Numerical Simulation of the Lux Vertical Axis Wind Turbine

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Mathematics and Statistics
University of Saskatchewan
Saskatoon

By
Fagbade Adeyemi I.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Mathematics and Statistics
142 McLean Hall
106 Wiggins Road
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5E6

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

Wind energy can be characterized as a cheap, clean, and renewable energy source that is absolutely sustainable. With increasing demand for wind energy, it is productive to investigate the structural and operational factors that undermine the proficiency and the characteristic performance of the wind turbine. Of paramount importance to efficient wind energy generation is the aerodynamics of the wind turbine blades. The aerodynamic factors, such as drag, airfoil profiles, and wake interactions that often reduce the performance of the wind turbines, can be investigated through computational mathematics using computational fluid dynamics (CFD). CFD offers basic techniques and tools for simulating physical processes and proffers important insights into the flow data, which are demanding and costly to measure experimentally.

In this thesis, we develop a simulation model in an open-source software package called OpenFOAM to investigate the performance characteristics of the Lux Vertical Axis Wind Turbine (VAWT). The Lux VAWT has a simpler design than its horizontal counterparts; however, its performance is affected by the unsteady aerodynamic due to a complex flow field. The turbulent flow field is governed by the incompressible Navier–Stokes equations. Simulations are carried out with an unsteady incompressible and dynamic flow solver, PimpleDyMFoam, on an unstructured mesh surface of the Lux VAWT geometry. The computational domain includes both the stationary and rotating mesh domains to accommodate the rotating motion of the turbine blades and the free-stream zone. The arbitrary mesh interface is applied as a boundary condition for the patches between the two domains to enable computation across disconnected but adjacent mesh domains. Meshing was done using two separate meshing tools, `snappyHexMesh` and ANSYS Mesher. The `snappyHexMesh` tool offered the most flexible and effective control over the mesh generation and quality. In order to derive the maximal power output from the Lux VAWT simulations, the Unsteady Reynolds-Averaged Navier–Stokes (URANS) equations are solved with different time-stepping methods; the objective is to reduce the computational costs. While attempting to reduce the numerical diffusion from the non-transient terms of URANS, a stabilized trapezoidal rule with a second-order backward differentiation formula (TR–BDF2) time-stepping method was implemented in OpenFOAM.

As a result, the transient aerodynamic forces of the blades, the torque, and power output are evaluated. The findings demonstrate that most of the transient aerodynamic force is generated along the axis of rotation of the rotor during one complete revolution. Similarly, the computations indicate that the BDF2 method results in the least computational cost and predicts a turbine power that is somewhat comparable to the experimental results. The difference between the simulation results and the experimental data is attributed partly to the pressure fluctuations on the turbine blades due to the mesh topology.

Acknowledgements

I would like to express my greatest appreciation to my academic advisor, Professor Raymond Spiteri, for his continuous interest, patient guidance, useful critiques and brilliant ideas during this study. I would also like to thank Dr. Reza Rhaghgoo for his untiring effort, insight and valuable assistance during the computation. I wish to acknowlegde the help provided by my colleagues in the Numerical Simulations Laboratory, and thank them all. I am truly grateful to my family for being tremendously patient and encouraging during the time I had to devote my attention to this work. Without them, I would never be able to tackle all the hardships associated with graduate school and studying abroad. Special thanks to Dayo for taking time to review this thesis, sentence by sentence, and giving insightful feedbacks. My friends who contributed to this work by their encouragement are owed my deepest gratitude.

This is dedicated to the one I love–Queen and Alexander. You are truly my values.

Contents

List of Tables

List of Figures

List of Abbreviations

CFD	Compuational Fluid Dynamics
TR–BDF2	Trapezoidal rule with second order Backward Differentiation Formula
DNS	Direct Numerical Simulation
LES	Large Eddy Simulation
RANS	Reynolds Average Navier–Stokes
URANS	Unsteady Reynolds Average Navier–Stokes
AFD	Analytical Fluid Dynamics
EFD	Experimental Fluid Dynamics
FDM	Finite Difference Method
FEM	Finite Element Method
FVM	Finite Volume method
N–S	Navier Stokes
TSR	Tip Speed Ratio
RPM	Revolution Per Minute
IBVP	Initial Boundary Value Problem
IC	Initial Condition
BC	Boundary Condition
MRF	Multiple Reference Frame

List of Symbols

$\mathbf{U}$	velocity vector
$d\mathbf{S}$	outward unit normal vector
V_P	control volume around point P in the domain
$\mathbf{x}$	position vector of the fluid particles in the domain
σ	stress tensor
e	total specific energy
Q	volume energy source
S	total specific entrophy
T	temperature
q	heat flux
p	fluid pressure
R	ideal gas constant
T	temperature
ρ	fluid density
q	heat flux
a_1	arbitrary constant
μ	dynamic viscosity of the fluid
F	total external body force
Ω_{ij}	rotation tensor
d	the distance from the field point to the nearest wall
ν	molecular kinematic viscosity
$\overline{S}$	magnitude of the mean vorticity
ω	specific dissipation rate or turbulent frequency
ϵ	dissipation rate
k	turbulent kinetic energy
σ_k	turbulent Prandtl number for k
σ_ω	turbulent Prandtl number for ω
μ	turbulent viscosity
F_1, F_2	blending functions within the flow regime
$CD_{k\omega}$	positive portion of cross–diffusion term
$\mathbf{S}_f$	face area vector
$S_{i,j}$	rate-of-strain tensor for the resolved flow.
D	rotor diameter

H rotor height

1 Introduction

1.1 Overview

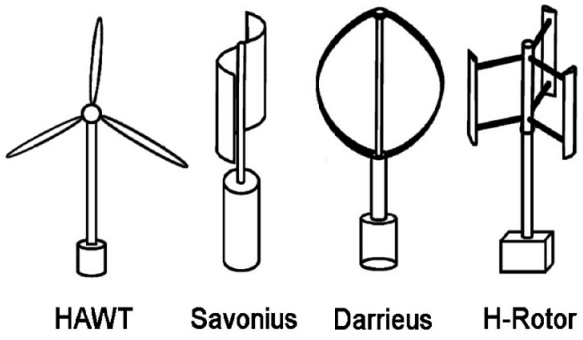
The increasing environmental impact posed by modern methods of power generation continues to be a source of concern for energy regulatory bodies across the globe [66]. For example, in a study conducted in 2009, the United States Department of Energy [125] reported that fossil fuel extraction alone constitutes 80% of recorded environmental degradations, in addition to the harmful effects of carbon-monoxide emissions that often deteriorate the environmental balance resulting in global warming. This impact has called for alternative sources of power generation, one of which is wind energy. Wind energy as a source of renewable energy is clean and environmentally friendly and contributes little or no greenhouse gases throughout the operational cycle [47]. The utilization of wind energy is technologically driven and has gained importance as one of the best alternatives to the traditional methods of generating electricity with a viable potential to mitigate and reduce global warming effects on the environment.

Wind power is a viable and renewable resource towards a balanced future: wind power and hydro power are among the most cost-effective and fastest-growing renewables in the world [95]. Wind energy is expanding at a rate of 30% per annum, and by the end of 2012, it was reported that the worldwide installed capacity of wind energy reached 282,482 megawatts (MW) [50] because of continued investment by governments and corporations. In Canada for instance, there is a renewed attention to wind power. In the year 2016 alone, 21 new wind energy projects contributed to a growth of 702 MW of wind generation. Currently, wind energy accounts for approximately six percent of Canada's electricity supply, producing adequate power to sustain over three million Canadian homes [44]. For many countries and their fast potentials, wind power can serve as a good starting point to expand sustainable energy source, although, due to unevenness, it may not be the singular source of electricity for a whole country. However, policies with incentives are being formulated worldwide to promote the significance of wind power in future energy supplies.

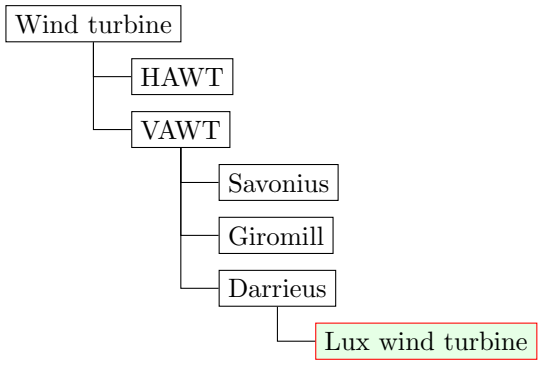
1.1.1 Wind Turbine Design

The history of harvesting the wind energy traces back to the tenth century in Persia, with the use of a mechanical device called the windmill [116]. The inhabitants of the Eastern Persia, who were predominantly farmers, used the windmill to generate mechanical power to mill grain and pump water for farming-related activities. However, in the late nineteenth century, there was a change from the windmills mechanical

functionality to the wind turbines generating electrical power. Windmills are similar to the wind turbines in that they both operate on the thrust exerted by the wind; however, wind turbines extract the kinetic energy in the wind and transform it into electricity. Understanding the wind energy conversion is simple. Irregular warming of the atmospheric surface by the sun results in intermittent wind flow. Wind turbines convert the kinetic energy from the wind flow into the mechanical energy, which, through alternators, is converted into electricity that is clean and inexpensive. Through innovative technologies, wind turbines have metamorphosed into an essential commercial avenue for large-scale power generation. The world's largest-capacity wind turbines to date are capable of delivering up to 9MW power. With continued investment and effective government policies with incentives, the use of wind turbines to harness wind energy seems a viable prospect. Many different designs of wind turbines have emerged over the years. However, contemporary wind turbines can be identified based on the shaft orientation and rotational axis as either the horizontal-axis variety or the vertical-axis design such as the Savonius type, the Darrieus type, or the Giromill type (see Figure 1.1). A turbine whose shaft is parallel to the ground is a horizontal-axis wind turbine (HAWT). In this design, the axis of rotation is in the same direction as the wind, and the turbine blades apply the aerodynamic lift to spin perpendicular to the direction of the wind. However, a vertical-axis wind turbine shaft is perpendicular to the ground as shown in Figure 1.2. The two models have their peculiar but distinct rotor designs, each with its own distinct performance and favorable characteristics [56]. Even though both designs have strengths and shortcomings, HAWTs account for the majority of the utility-scale projects due to their higher efficiency under consistent wind conditions [114].



(a) Schematic of Horizontal-Axis, Savonius drag-based, Darrieus curved-blade,and Giromill H-Rotor wind turbines [36].



(b) Categories of wind turbines.

Figure 1.1: Type of wind turbines.

The vertical-axis wind turbines (VAWTs), illustrated in Figure 1.3, have the main rotor shaft assembled perpendicularly and the primary components (such as the generator) positioned near the bottom of the turbine for easy mounting and repair. They are built to capture the wind kinetic energy irrespective of wind

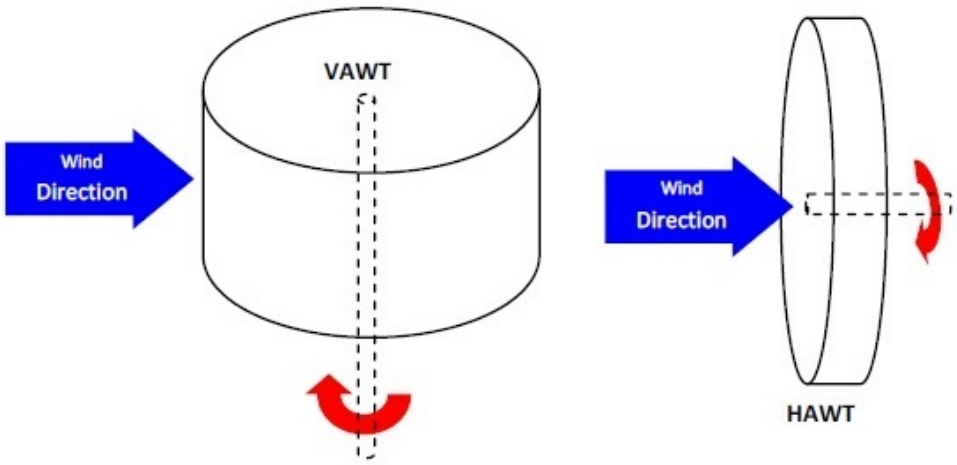


Figure 1.2: Turbine shaft configuration and rotor orientation [28].

Figure 1.3: Typical example of VAWT design (VAWT Texas A & M)

orientation, thus, setting aside the need for re-positioning along the wind and thereby offering great benefits in places where the wind direction keeps varying. VAWTs are characterized by lower aerodynamics noise and fit in more readily into urban environments [92]. Unlike the HAWT, where the gearbox and alternator are located at the top of the tower, VAWTs feature a direct drive where the alternator and gearbox are generally placed near the base. This positioning significantly minimizes operating costs while increasing durability and reliability [122]. In addition, VAWTs have distinct operating features such as the ability to operate under irregular wind flow, slow cut-in speed, and low maintenance cost [35]. With these advantages (refer to Table 1.1), VAWTs have the potential of becoming the main technology for urban wind generation due to the slower, more turbulent, multi-directional characteristics of wind in most cities [98]. However, there are some limitations to the use of VAWTs, namely the low power output and the fluctuating aerodynamic forces on the turbine blades that can pose a significant fatigue problem for the entire turbine system [89]. Nonetheless, VAWTs have predominantly been used in small-scale businesses especially for domestic purposes and are attracting growing interest globally.

Table 1.1: Difference between HAWTs and VAWTs. Retrieved from www.windturbinestar.com

Wind turbines performance chart			
Nos	**Performance**	**Horizontal–axis design**	**Vertical–axis design**
1	Power generation efficiency	50%–60%	Above 70%
2	Electromagnetic interference	Yes	No
3	Steering mechanism of the wind	Yes	No
4	Gear box	Above 10 kW:Yes	No
5	Blade rotation space	Quite large	Quite small
6	wind-resistance capacity	Weak	Strong (can withstand typhoon up to class 12–14)
7	Noise	5–60 dB	0–10 dB
8	Starting wind speed	High (2.5 m/s – 5 m/s)	Low (1.5 m/s – 3 m/s)
9	Ground projection effects on human beings	Dizziness	No effect
10	Failure rate	High	Low
11	Maintenance	Complicated	Convenient
12	Rotating speed	High	Low
13	Effect on birds	Great	Small
14	Cable stranding problem	Yes	No
15	Power curve	Depressed	Full

In distinction to the earlier classification of wind turbines, they can again be categorized based on the dominant aerodynamic forces used to generate power. This includes the Drag-type and Lift-type turbines [69]. This classification is certainly a minor subdivision because the turbine design experiences both the drag and lift forces in different proportions. The two forces influence the performance characteristics of modern

wind turbines such as drag VAWT shown in Figure 1.1a, or a situation where one force undermines the turbine performance (such as the drag force on the H-rotor VAWT). HAWTs are generally lift-type and operate identically to the wings of an airplane, where the lift component of the aerodynamic force is used to generate power. VAWTs, however, have more diversity and are commercially available as Savonius, Darrieus, and Giromill turbines (see Figure 1.1b).

The simplest among these VAWTs is the Savonius drag-driven turbine, designed by Sigurd Savonius in the 1920s. The Savonius drag-driven turbine was designed intentionally for use in sailing, water pumping, and ventilation using air and water. However, recent advancements in the rotor design have facilitated its use in power generation. Generally, the Savonius turbine can be described as a drum cut vertically into two halves. The two parts are fixed to the corresponding sides of the vertical shaft forming scoops to capture the wind energy and work entirely on the thrust force of the wind to rotate the generator. The performance of the Savonius rotor is built upon the difference of the drag force when the wind strikes the concave and convex parts of the semi-spherical blades [96]. The Savonius turbines are characterized by minimal noise, low wind speeds (cut in speeds), independent of the wind direction, and simplicity of manufacture and operation [109]. However, the Savonius rotor has a low power coefficient when compared to HAWTs. Nevertheless, Zamani et al. [135] remarked that the main argument in supporting drag-based turbines is the self-start ability, in contrast to other types of VAWT, and their relatively low cost [91] compared to lift-based turbines. As a result, there are cost benefits to employing this turbine technology.

The lift-driven (Darrieus) VAWT as shown in Figure 1.1a was developed by Sandia National Laboratories in the USA in the 1980s and has been extensively redesigned for optimal performance [57]. The Darrieus VAWTs operate through aerodynamic lift and have two or more airfoil blades attached to the main vertical shaft. However, a typical Darrieus-style vertical–axis wind turbine shown in Figure 1.4 has curved blades (airfoils) which fly through the wind on their power strokes as they rotate around the shaft. When the Darrieus rotor spins, the blades (airfoils) rotate. The rotor rotates at a different angle to the wind speed, and usually many times faster, sufficiently enough to power the generator. The aerodynamics principles which rotate the rotor are the same as helicopters in rotorcraft designs. Although Darrieus turbines have low starting torque and generally suffer from poor building installation and integration [33], they have proven to be more efficient than Savonius–type turbines and have better performance with higher tip speed ratio [124].

The H-Rotor turbines otherwise called Giromill turbines are similar to the Darrieus turbines but have straight blades as opposed to curved ones. The H-rotor turbines are characteristically designed with lower speed ratio and high self-starting capacity. They are characterized as having a relatively simple structure, variable pitch mechanism that reduces the torque pulsating, and higher aerodynamic performance than the curved blade turbines or beater type VAWTs [15]. However, findings revealed that turbines that use primarily lift forces (Darrieus design) have better and higher efficiency than drag force designs [23].

Even though VAWTs have a simpler design, their performance characteristics are limited by many complications primarily attributed to the complex flow-field around and within the rotor. The analysis of VAWT

Figure 1.4: Darrieus vertical wind turbine [89].

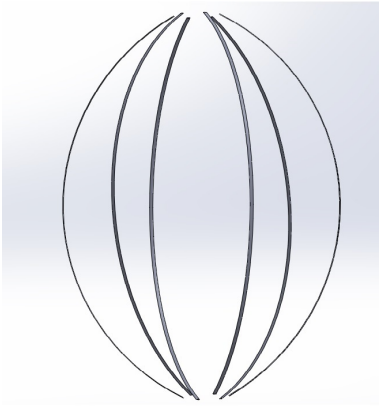
aerodynamics presents a number of factors yet to be investigated, and several design factors, such as blade shape configuration and tip speed ratio, are yet to be optimized. Understanding these fundamental aerodynamic characteristics of the VAWTs is vital for efficient design, operation, and performance of wind turbines. For example, in a study conducted by Saha et al. [109], a wind tunnel experiment is performed to determine the aerodynamic performance of the Savonius rotor systems with different blade shapes (semicircular and twisted blades) on different rotor-stages. The experiments examined the characteristic impact of different design factors such as rotor-stage number, number of blades, and blade shape on the turbine performance characteristics, and observed that the two-stage rotor demonstrated a favorable performance characteristic when matched with the three-stage rotor. They also observed that as the number of rotor stages increased, the inertia on the rotor increased, causing a dramatic reduction to the turbine output power. However, the performance of the twisted blade rotor in a two-bladed system was found to be better than the semicircular-bladed rotor. Similarly, Mosfequr et al. [96] in 2010 carried out an experiment on the drag and torque characteristic of the three-bladed Savonius turbine system with and without overlapping blades. They remarked that the Savonius turbine system without overlapping blades has a better drag and torque characteristic and optimize the aerodynamic performance of the turbine even in the presence of high Reynolds number. Recent comparative experiments by Gupta et al. [49] showed that a hybrid Savonius–Darrieus turbine can boost the aerodynamic characteristic of the hybrid turbine and achieve a power coefficient of about 0.51. The comparative study suggests that there is a measurable gain in the power coefficient of the coupled Savonius–Darrieus rotor without overlap. In contrast to the autonomous Savonius design, a hybrid Savonius–Darrieus turbine without overlapping offers better performance within the Savonius rotors [16]. Gupta et al. remarked that the hybrid turbines showed a better performance than the totality of any singular turbines because of the short startup time in the hybrid design. Although the aerodynamic flow interference between the hybrid rotors affect the power output, the influence is summarily overcome by the benefits of improved startup [49]. This benefit is due to the Savonius's high solidity and torque, combined with the high tip-speed ratio of

a Darrieus design. This illustrates ample usefulness of the characteristic and operating advantages of an individual turbine [9]. Such operating advantages are key for any wind turbine designed for use in an urban environment.

1.1.2 The Lux VAWT

The Lux VAWT is a promising model of vertical-axis wind turbine, which can be used in urban environments where HAWTs are constrained. It is a novel wind turbine, invented by Glen Lux in 2013 with the sole purpose of generating electricity for homes and industries. The Lux VAWT is a modified version of the Darrieus design-type with six curved blades like the egg-beater type shown in Figure 1.5a. The modular and innovative design is quiet, structurally fascinating, and convenient for residences and institutions. Like other VAWTs, the Lux VAWT design concept has a direct drive train, and its major components are situated near the base for easy accessibility and maintenance. The Lux VAWT installation reduces the amount of land required by up to 90%, requires significantly less material, and costs much less than the contemporary wind turbines [126]. The Lux VAWT prototype currently generates approximately 50 kW of electricity. At full capacity, a typical commercial wind turbine connected to a power grid can generate 600 kW–2.0 MW [47] of electricity. In order to make the Lux VAWT more commercially viable, a larger version must be developed. The primary goal of this research is to numerically investigate the performance characteristics of the Lux VAWT geometry, schematically represented in Figure 1.5b, through CFD simulations. Aerodynamic phenomena such as flow separation, unsteady downstream wake, and complex flow over the blade areas, must be properly investigated to optimize the Lux VAWT performance.

(a) Prototype of the Lux Turbine [42].



(b) Lux wind turbine geometry.

Figure 1.5: The Lux Turbine

1.2 Computational Fluid Dynamics

Fluid (gas or liquid) flows are primarily modeled by a system of differential equations that describe the conservation laws for mass, momentum, and energy [130]. These equations, particularly the Navier–Stokes equations, are quite difficult to solve analytically; however, computational fluid dynamics (CFD) offers a numerical approach. CFD, as a branch of fluid mechanics, applies numerical analysis and data assembly to obtain an approximate solution to the problems of fluid dynamics and heat transfer [39]. It facilitates a qualitative analysis in addition to a quantitative prediction of fluid flow through mathematical modeling (using theory of differential equations), numerical methods, computing packages, and tools (Pre–processing, Solvers, and Post–processing utilities; see Table 1.2). CFD simulations have become an important part of applied projects because they provide basic methods and tools, illustrated in Table 1.3, for simulating physical processes and reducing experimental cost [135]. Over the years, computer simulations of the Navier–Stokes equations have been adapted to deal with wind turbines problems because CFD is seen as a key contributor in designing, analyzing, monitoring, and virtual prototyping of everything involving wind turbine development [130]. Now, there is a possibility of simulating the turbulent flow past a wind turbine with high-performance CFD codes. These codes apply different turbulence models to simulate the atmospheric turbulence effects on the performance of wind turbines. However, there are still challenges with some of the CFD techniques and codes to resolve all the complex flow phenomena in wind turbines aerodynamics.

Steps	Activities
Pre–processing	• Geometry generation • Geometry set up • Meshing of the Geometry
Solver	• Problem specification • Additional models • Numerical computation
Post–processing	• Line and Contour Data • Average Values • Report Generation

Table 1.2: Methodology in CFD.

1.	**Problem description**	stating relevant conditions of flow
2.	**Mathematical model**	IBVP = PDE + IC + BC
3.	**Mesh generation**	grids, nodes/cells size, time instants
4.	**Space discretization**	control volume, coupled ODE system
5.	**Time integration**	algebraic system $\mathbf{Ax} = \mathbf{b}$
6.	**Interative solver**	discrete function values
7.	**CFD software**	implementation, troubleshooting
8.	**Simulation run**	control parameters, convergence conditions
9.	**Post–processing**	flow visualization, data analysis
10.	**Verification**	model validation/adjustment

Table 1.3: CFD analysis process.

Over the years, CFD researchers have applied several basic techniques to study the aerodynamic characteristics of the VAWTs and their inherently unsteady flow fields. Operational factors, such as computation cost, investment in time, and accuracy have always been the guiding principles to determine the appropriateness of a given technique especially when the wind turbines are placed in a turbulent flow field. For instance, CFD techniques such as the actuator disk method have been applied to simulate wind turbines aerodynamic and perform sensitivity analysis on the turbine system [5]; however, numerical simulations with recent CFD packages, such as Ansys Fluent, OpenFOAM [100], etc., have the capacity to offer better insights into the wind turbines aerodynamics and provide data that are either difficult to measure or test experimentally.

Some CFD methods centered on the numerical solution of the Reynolds-Averaged Navier–Stokes (RANS) equations with accompanying turbulence models [119]. This thesis, however, focuses on the application of mesh-based methods. Mesh-based methods discretize the computational domain into small control volumes, called cells, and solve the Navier–Stokes equations on each cell at a specific time. These methods require a tremendous amount of computer time and memory especially when the unsteady and viscous effects are significant.

Mesh-based methods consist of the Finite Difference Method (FDM) [106], Finite Element Method (FEM), and Finite Volume Method (FVM) [105]. In the last few years, the FVM has become more popular and

dominant in CFD applications to analyze the flow behaviors of wind turbines than the other two, due largely to the accurate and minimal cost of computation [130]. Full CFD simulations of a wind turbine have already been accomplished, and researchers were able to match the numerical results to the experimentally measured data. Ferreira et al. [38] used the FVM to simulate dynamic stall in a VAWT and compared the results with experimental data. They also made a comparison in terms of accuracy of the frequently applied turbulence models such as Unsteady Reynolds-Averaged Navier–Stokes (URANS) Spalart–Allmaras and k–ϵ, Large Eddy Simulation (LES) and Detached Eddy Simulation (DES). Castelli et al. [22] simulated a low solidity vertical-axis micro wind turbine at a temperate tip speed ratio with the URANS k–ϵ turbulence model. They statistically inferred that the accuracy of the CFD simulations can be increased through mesh refinement especially within the turbulent boundary layer. Castelli et al. also compared their results with the wind tunnel study conducted at the Politecnico di Milano [12]. Although the wind tunnel data were approximately compared to the numerical results, their experimentally measured data could only be considered as a rough estimate because the tunnel blockage effects were neglected in the experiments. Howell et al. [61] conducted a wind tunnel study along with a two- and a three-dimensional numerical study of a small VAWT and examined the influence of moderate wind speed, tip speed ratio, and blade solidity on the performance characteristics of the turbine. They proposed that the turbine's performance is contingent on the number of blades and their surface roughness. However, there were some discrepancies between the two-dimensional numerical results and the experimentally measured data. These discrepancies were associated with the failure of the two-dimensional model to simulate the large tip vortices of the flow. Similarly, Brahimi et al. [19] examined the validity and accuracy of several aerodynamic analysis models, such as stream-tube models and numerical simulations, with experimental data. They demonstrated that the dynamic stall models accurately estimate the aerodynamic loads and the performance characteristics of Darrieus wind turbines. The models are, however, constrained to the specific airfoil type and the flow field applied in the experiment.

Abdullah et al. [24] conducted a rigorous parametric study on the aerodynamic characteristics of VAWT in a tilted condition. The CFD simulations of the VAWT were performed by solving the unsteady RANS equations using an open-source package, OpenFOAM. They [24] implied that the power coefficient of the VAWT is directly influenced by the upright position of the turbine, especially the tilted angle of the blade configuration. They maintained that tilted condition less than three degree slope is likely to improve the performance characteristics of the VAWT. Also, Orlandi et al. [101] examined the influence of skewed winds on the aerodynamic characteristics of an inclined H-type VAWT using blade element-momentum (BEM) and CFD approaches. The CFD results were numerically corroborated with the experimentally measured data of a full-scaled Darrieus VAWT. The CFD simulations indicated that the power gain in the tilted flows increased during the downward posture of the turbine revolution. The CFD investigation concludes that the performance deterioration as a result of the skewed flow is generally low, and the power coefficient may be enhanced. Similarly, Abdolrahim et al. [108] conducted a sensitivity analysis on the performance characteristics of 2-bladed H-type low-solidity VAWT operating at a moderated tip speed ratio. The 3D

unsteady RANS simulations were performed with the 4-equation transition turbulence model. The study emphasized the importance of the sensitivity analysis on the accurate prediction of VAWT performance. The paper concludes that sufficient azimuthal increment and fine grid resolution are important sensitive parameters to accurately measure the VAWT power generation.

Therefore, in light of the above, this thesis seeks to create a simulation model with CFD tools to measure the performance characteristics of an urban-scale Lux VAWT. The thesis goals are to get a deeper understanding on how to optimize the run-time of the CFD codes in parallel computing and run the CFD simulations to match the experimentally measured data particularly when turbulent and viscous flow are of great influence. The modular and operational characteristics of the Lux VAWT have been selected to streamline the flow physics and expedite the simulation procedures. As a result, the thesis approach is centered on the prediction of the flow phenomena. The long-term goal is to make the CFD simulations results as credible as the experimentally measured data when considering design options and make CFD the commercial de-facto standard during the initial design of the Lux VAWT.

1.3 Fundamentals of Vertical-Axis Wind Turbines

An efficient design of a vertical-axis wind turbine depends on both the structural and the aerodynamic characteristics of the turbine system [109]. Understanding VAWT aerodynamics, especially the dynamics flow of air over the airfoil, helps in designing, modeling, and evaluating the wind turbine performance characteristics. Because the focus of this thesis is to numerically investigate the performance characteristics of the Lux VAWT, it is worthwhile to illustrate the basic aerodynamic forces acting on the turbine blades.
Figure 1.6 illustrates two substantial forces on a symmetric airfoil when wind flows over it; a lift force that is normal to the wind and a drag force in a direction of the wind [31]. Wind flows over the surface of the airfoil at a different angles of attack (i.e., the angle between the incoming flow and the chord of the airfoil) and follows the shape of the airfoil; if the airfoil is curved, the wind is driven downward by the airfoil because the air flows more rapidly over the top of the airfoil and less under the airfoil [88]. At zero angle of attack the symmetric airfoil generates zero lift; however, the cambered airfoil creates positive lift force. Fundamentally, any turbine blade characterized by symmetric airfoil generates no lift force at zero angle of attack because the net force at the top section of the blades instantly nullifies the net force at the bottom [121].

The existence of the lift force depends not only on the angle of attack and the velocity of the wind around the airfoil but also on the type of flow regime over the airfoil. For instance, in a separated flow (where a flow is no longer following the airfoil contour), an increase in the angle of attack increases the separation area near the leading edge up to the stall condition. The separation, however, causes a drastic reduction in the generated lift due to an increase in the pressure drag over the airfoil. The lift force is generated by the difference of the pressure distribution between the upper and lower surfaces of the airfoils as they sweep

through the flow [53].

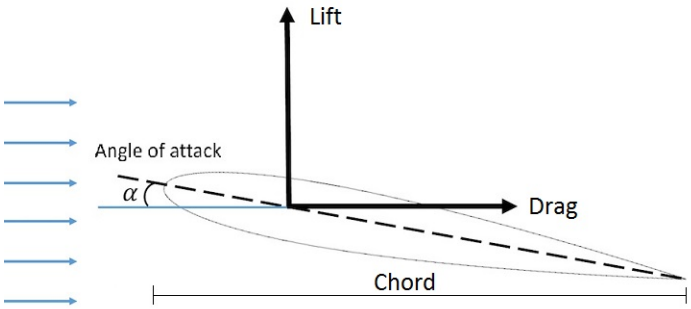


Figure 1.6: Lift and drag forces on a symmetric airfoil (https://www.comsol.com/blogs/how-do-i-compute-lift-and-drag/)

In the same manner, the wind blowing over the airfoil surfaces produces a notable drag force along the wind direction. The drag force is a result of the viscous forces of the flow and the fluctuating pressure distribution over the airfoil areas. It is a loss term that must be minimized in order to produce high-performance wind turbine systems.

Quantitatively, these aerodynamic forces can be determined using the angle of attack of the air flow over the airfoil and the perceived velocity W (air velocity relative to the airfoil). When a VAWT operates in a rotating domain, the perceived velocity and angle of attack change intermittently [63].

Considering Figure 1.7, let θ be the azimuthal angle of the airfoil, R the radius of the turbine, and ω the rotational velocity.

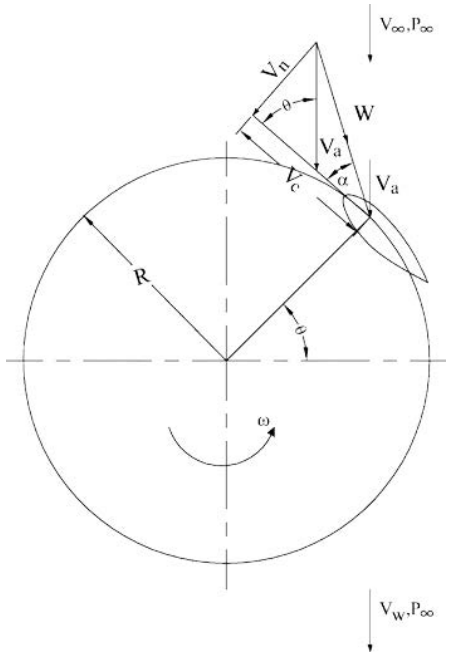


Figure 1.7: Inflow condition in a rotating frame [63].

The inlet wind speed can be resolved into radial and tangential components, using the azimuthal angle of the flow. The chordal velocity V_c and the normal velocity V_n are expressed by

$$V_c = R\omega + V_a \cos\theta,$$

$$V_n = V_a \sin\theta,$$

where V_a denotes the axial velocity through the rotor. With these velocity components, it is possible to determine the angle of attack. Meanwhile, when the turbine (rotor) is rotating, the airfoil moves through the air envelop in a circular path, creating a curved flow so that every location along the airfoil experience intermittent changes in the angle of attack. This angle of attack [31] can be determined by

$$\alpha = \arctan\frac{V_n}{V_c} = \arctan\frac{\sin\theta}{(\frac{R\omega}{V_a} + \cos\theta)}.$$

Accordingly, changes in the direction of the airfoil produce angles of attack alternating between the positive and negative maximal.

Then, the resultant of the oncoming velocity vector and the blade's induced velocity vector that creates this varying angle of attack is called the relative velocity [63], and is given as

$$W = \sqrt{V_c^2 + V_n^2}.$$

Having determined W and α , we can evaluate the magnitude of lift and drag forces [89] by

$$F_l = \frac{1}{2}\rho\, C_l C H W^2,$$
$$F_d = \frac{1}{2}\rho\, C_d C H W^2,$$

where C_t and C_n are the tangential and normal force coefficient [4] defined respectively as

$$C_t = C_l \sin\alpha - C_d \cos\alpha,$$

$$C_n = C_l \cos\alpha + C_d \sin\alpha,$$

and ρ is the air density, C is the chord length, and H is the height of the turbine [63].

Reynolds number

The Reynolds number is a dimensionless number that measures the ratio of the inertia forces to the viscous forces and describes the characteristic of the fluid flow conditions [3]. That is, by using the Reynolds number, one can quantitatively describe the degree of laminar flow or turbulent flow. For a high-speed wind turbine in an airflow, Reynolds number is determined by

$$Re = \frac{\text{Inertia forces}}{\text{Viscous forces}} = \frac{\rho U L}{\mu} = \frac{U L}{\nu}, \tag{1.1}$$

where μ is the dynamic viscosity, $\nu = \frac{\mu}{\rho}$ is the kinematic viscosity, and L is the characteristic length scale. The characteristic length can assume different values based on the geometry of the flow and the section that

interacts with the fluid. However, for most VAWT the characteristic length scale is commonly determined by the chord length of the airfoil (blade).

Rotor Swept Area

The rotor swept area denotes an area covered by the plane of wind intersected by the rotor. It is one of the significant variables in evaluating the power generated by the wind turbines. For most vertical-axis wind turbines, the swept area can be evaluated by

$$A_R = DH.$$

Blade Aspect Ratio

The blade aspect ratio expresses the ratio of the turbine blade length to its chord length. It is evaluated as

$$\mu_R = \frac{H}{C}.$$

Turbine solidity factor

The turbine solidity factor is a structural parameter [78] that expresses the ratio of the rotor area to the total swept area and can be calculated as

$$\sigma = \frac{nC}{D},$$

where n is the total number of blades on the rotor, C is the chord length, and D is the rotor diameter. An increase in the chord length and the solidity factor usually improves the turbine aerodynamic forces and consequently the maximum power output. A wind turbine with low solidity (say 0.10) has high speed but low torque output, while high solidity turbine (> 0.8) is characterized with low speed and high torque output.

Tip Speed Ratio

Another aerodynamic parameter relevant to the power output of the wind turbines is the tip speed ratio (TSR), which can be expressed as the ratio of the rotor speed to the oncoming wind speed. It is mathematically given as

$$\lambda = \frac{R_m \omega_m}{V_\infty},$$

where ω_m is the angular velocity of the rotor, R_m is the peak radius of the rotating turbine, and V_∞ is the unperturbed wind speed. The blade angular velocity in radians per second is obtained from revolutions per minute (rpm) of the rotor using the relationship

$$w_m = \frac{2\pi(rev)}{60}. \tag{1.2}$$

Power and Power Coefficient

For the airflow through a cylindrical column, the wind power can be expressed as

$$P_W = \frac{1}{2}\rho A_R V_\infty^3.$$

The power generated by a given wind turbine is derived from the kinetic energy of the wind as it flows over the entire system. Hence, the wind energy cannot be fully converted to mechanical energy by the rotor blades due to some aerodynamic loss and the atmospheric boundary layer effects; however, the instantaneous power output of the rotor is related to the estimated frontal area of the blades and is derived by multiplying the averaged torque of all the rotor blades with the turbine's angular velocity (ω)

$$P_T = \omega T_y, \tag{1.3}$$

where T_y is the component of the torque vector $\mathbf{T}$ along y-axis. The total resistant torque vector $\mathbf{T}$ [36] can be evaluated by adding the cross products of the pressure force $\mathbf{F}_p$ and viscous force $\mathbf{F}_v$ for each cell i and the vector $\mathbf{R}_i$, defining the distance of each computational cell center from the turbine axis of rotation

$$\mathbf{T} = \sum_i (\mathbf{R}_i \times \mathbf{F}_p + \mathbf{R}_i \times \mathbf{F}_v) \equiv (T_x, T_y, T_z). \tag{1.4}$$

The pressure force and viscous force vectors are determined by

$$\mathbf{F}_p = p_i \mathbf{A}_i, \tag{1.5}$$

$$\mathbf{F}_v = \nu_i \frac{\partial \mathbf{U}_i}{\partial \mathbf{x}_i} \mathbf{A}_i, \tag{1.6}$$

with subscript i denoting the cell i, p_i representing the total pressure in the cell i, and $\mathbf{A}_i$ is the total area of the vector surface element i (a vector normal to the surface) of the computational mesh. Similarly, the torque coefficient is calculated by

$$C_i = \frac{2\mathbf{T}}{\rho \mathbf{A}_i W^2 \mathbf{R}_i}.$$

It is important to note that wind turbines vary in size, blade aspect ratio and solidity, and power output. However, the variation can be normalized using a distinct parameter, called the power coefficient, otherwise known as the turbine efficiency. The power coefficient expresses the ratio of the derivable power P_T to the kinetic power P_W available in the undisturbed wind flow

$$C_P = \frac{P_T}{P_W} = \frac{P_T}{0.5\rho A_R W^3}. \tag{1.7}$$

The power coefficient has the theoretical peak value of 0.5926 when the ratio of the downstream to the upstream wind velocity is one-third. This maximum power factor is designated as the Betz limit [14], which represents the peak amount of power that can be extracted by a wind turbine from the available wind kinetic energy. The turbine efficiency C_P helps in the comparison of one turbine system with others and determines whether the use of the turbine will provide the expected power output.

1.4 Thesis Contributions

This research work investigates the modeling and the aerodynamic simulation of the Lux VAWT system via computational fluid dynamics with an open-source software package called OpenFOAM. The unsteady turbulent flow is simulated under various inlet wind conditions with the OpenFOAM solver for unsteady flow (pimpleDyMFoam). The CFD simulations were performed with different temporal discretization schemes to predict the flow behaviors over the Lux VAWT blades and produce simulation data that fairly agree with the experimentally measured data on the Lux VAWT. In achieving these goals, we have contributed to the aerodynamic simulations of the Lux VAWT system by:

- designing a computational geometry of the Lux VAWT using 3-dimensional CAD software SolidWorks and ANSYS,
- generating computational meshes (of different sizes) for the Lux VAWT using ANSYS and OpenFOAM,
- implementing a modified TR-BDF2 method in OpenFOAM,
- calculating the averaged power output of the Lux VAWT using the k-ω SST turbulence model and compare the CFD simulation results with experimentally measured data.

1.5 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 summarizes the governing equations of continuum mechanics together with the constitutive relations for Newtonian fluids. A short overview of turbulence modeling and discretization techniques pertaining to the CFD is also presented. Discretization of a general scalar transport equation is examined term by term in a control volume of the problem domain. Chapter 3 outlines OpenFOAM's capability, utility, and procedures for mesh generation and numerical simulation. Chapter 4 presents a detailed description of the numerical simulations along with the computational findings in comparison with the experimentally measured data. Finally, the last chapter includes conclusions from the current study and suggestions for future research directions towards improving the performance characteristics of the Lux VAWT.

2 Theoretical Background

2.1 Governing Equations

In this section, we describe the mathematical basis of fluid flow equations from the fundamental principles of conservation of mass, momentum, and energy, followed by a review of various methodologies on modeling turbulence and discretization schemes for the Navier–Stokes equations.

2.1.1 Fundamental Equations of Continuum Mechanics

Fluid flow is a typical example of continuum mechanics problems often encountered in nature. In these problems, the actual scales of the individual fluid particles (matter) are substantially smaller than the characteristic length and time scales of the flow. As a result, the macroscopic physical properties of the flow can be described by a continuous function in macroscopic coordinates of time and space [68]. Using the material derivative approach, we can express the rate of change of a given physical property ϕ in time by

$$\frac{d}{dt}\left(\int_{V_{\mathtt{P}}} \rho\phi \,\mathrm{d}V\right) = \int_{V_{\mathtt{P}}} \frac{\partial}{\partial t}\left(\rho\phi\right)\mathrm{d}V + \oint_{\partial V_{\mathtt{P}}} \mathrm{d}\mathbf{S}\cdot(\rho\phi\mathbf{U}), \tag{2.1}$$

where $\mathbf{U}$ represents the fluid velocity vector, $V_{\mathtt{P}}$ is the control volume at point $\mathtt{P}$ delimited by the boundary surface $\mathrm{d}\mathbf{S}$, and normal to $\partial V_{\mathtt{P}}$. The rate of change of ϕ in $V_{\mathtt{P}}$ is equal to its volume and surface sources

$$\frac{\partial}{\partial t}\left(\int_{V_{\mathtt{P}}} \rho\phi \,\mathrm{d}V\right) + \oint_{\partial V_{\mathtt{P}}} \mathrm{d}\mathbf{S}\cdot(\rho\phi\mathbf{U}) = \left(\int_{V_{\mathtt{P}}} Q_{V(\phi)}\,\mathrm{d}V\right) + \oint_{\partial V_{\mathtt{P}}} \mathrm{d}\mathbf{S}\cdot\mathbf{Q}_S(\phi), \tag{2.2}$$

The above equation can be written in differential form without reference to a particular volume as

$$\frac{\partial}{\partial t}\left(\rho\phi\right) + \nabla\cdot(\rho\phi\mathbf{U}) = Q_\phi, \tag{2.3}$$

where $Q_\phi = Q_V(\phi) + \nabla\cdot\mathbf{Q}_S(\phi)$ is the generic source of the physical property ϕ. Generally, the fundamental equations of any problem in continuum mechanics [16] can be written in the form

- Conservation of mass

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{U}) = 0, \tag{2.4}$$

- Linear momentum equation [39]

$$\frac{\partial}{\partial t}\left(\rho\mathbf{U}\right) + \nabla\cdot(\rho\mathbf{P}) = -\nabla p + \mathbf{f} + \nabla\cdot\sigma. \tag{2.5}$$

Note that $\mathbf{f} = \rho\mathbf{g}$ accounts for the external forces applied to the fluid and $\mathbf{P} = \mathbf{UU}$ where $P_{ij} = U_i U_j$ is a 3×3 matrix. Then

$$\nabla \cdot (\rho\mathbf{P}) = \rho \begin{bmatrix} \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} \end{bmatrix} \cdot \mathbf{P} = \rho \begin{bmatrix} \frac{\partial P_{11}}{\partial x_1} + \frac{\partial P_{21}}{\partial x_2} + \frac{\partial P_{31}}{\partial x_3} \\ \frac{\partial P_{12}}{\partial x_1} + \frac{\partial P_{22}}{\partial x_2} + \frac{\partial P_{32}}{\partial x_3} \\ \frac{\partial P_{13}}{\partial x_1} + \frac{\partial P_{23}}{\partial x_2} + \frac{\partial P_{33}}{\partial x_3} \end{bmatrix}^T$$

and the stress tensor

$$\sigma = \sigma_{ij} = (-p + 2\beta\nabla \cdot \mathbf{U})\,\delta_{ij} + \tau_{ij},$$

where β is the transport coefficient representing the volume viscosity and δ_{ij} is the Kronecker delta

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

τ_{ij} is the viscous stress tensor characteristically defined by the fluid type.

- Energy equation (first law of thermodynamics)

$$\frac{\partial}{\partial t}(\rho e) + \nabla \cdot (\rho e\mathbf{U}) = \rho\mathbf{g} \cdot \mathbf{U} + \nabla \cdot (\sigma \cdot \mathbf{U}) - \nabla \cdot \mathbf{q} + \rho Q, \tag{2.6}$$

where e is the total specific energy (including internal, potential, and kinetic energies), $\mathbf{q} = -k\nabla T$ is the heat flux vector relating temperature gradient via the Fourier law of heat condition and k is the thermal conductivity.

These transport equations are valid for any continuum and serve as the basis of CFD simulation.

2.1.2 The Navier–Stokes Equations and Boundary Conditions

The mathematical equations describing the motion of fluid particles are the Navier–Stokes equations [11] and are derived as direct consequence of Newton's law of motion for fluid. For the case of a compressible Newtonian fluid, the governing equations comprise of five partial differential equations (PDEs) [81]. The first equation represents the continuity equation, derived from the principle of conservation of mass, while the next three describe the conservation of linear momentum in three dimension (3D). The last is the energy equation obtained from the principle of conservation of energy (i.e., first law of thermodynamics). These equations are represented in the conservative form

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho\mathbf{U}) = 0, \tag{2.7}$$

$$\frac{\partial}{\partial t}(\rho\mathbf{U}) + \nabla \cdot (\rho\mathbf{P}) - \nabla \cdot (\mu\mathbf{K}) = -\nabla p + \rho\mathbf{g}, \tag{2.8}$$

$$\frac{\partial}{\partial t}(\rho e) + \nabla \cdot (\rho e\mathbf{U}) = \rho\mathbf{g} \cdot \mathbf{U} + \nabla \cdot (\mu\mathbf{W}) - \nabla \cdot \mathbf{q} + \rho Q, \text{ in } \Omega \subseteq \mathbb{R}^3,\ t > 0. \tag{2.9}$$

Note that $\mathbf{U} = (U_1, U_2, U_3)^T$, $\nabla = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3}\right)$, $\mathbf{W} = \mathbf{UK}$ ($W_{i,j} = U_i K_j$), and

$$\mathbf{K} = \nabla \mathbf{U} = \begin{pmatrix} \frac{\partial U_1}{\partial x_1} & \frac{\partial U_2}{\partial x_1} & \frac{\partial U_3}{\partial x_1} \\ \frac{\partial U_1}{\partial x_2} & \frac{\partial U_2}{\partial x_2} & \frac{\partial U_3}{\partial x_2} \\ \frac{\partial U_1}{\partial x_3} & \frac{\partial U_2}{\partial x_3} & \frac{\partial U_3}{\partial x_3} \end{pmatrix}$$

Unfortunately, this is an under-determined system because the number of unknown quantities ($p, \rho, \mathbf{U}$, and e) is greater than the number of equations in the system and is difficult to solve following an analytic approach. Hence, additional equations are required to describe a possible constitutive relationship between the fluid particles as they move relative to each other in order to close the governing equations.

If we apply the constitutive relationship relating pressure to flow density, we can obtain a closed system of equations. This relationship is seen in the general gas equation as follows

$$p = \rho R T,$$

where p is the flow pressure, T is the absolute temperature, and R is the specific gas constant. In this work, we assume air behaves like an ideal gas. With the equation of state, the flow density is linked to the pressure term in the momentum equations in the chosen pressure-based solver. The pressure-based solvers such as OpenFOAM solver–PimpleDyMFoam (see section 3.3) can switch between incompressible and compressible flow based on local Mach number. If the value of density is declared constant and set in the code, the code solves for incompressible flow.

In addition, the Navier–Stokes equations can be simplified using an assumption of an incompressible flow with a Mach number less than 0.3 ([104], [115]). The incompressible flow assumption is that for a fixed control volume the variation of density due to pressure changes is negligible, and the divergence of the velocity becomes zero [130]. Thus, for an incompressible, continuum, and isothermal flow, the Navier–Stokes equations can be rewritten as

$$\nabla \cdot \mathbf{U} = 0, \tag{2.10}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{P} - \nabla \cdot (\nu_t \mathbf{K}) = \mathbf{g} - \frac{1}{\rho} \nabla p, \tag{2.11}$$

or in component form

$$\begin{cases} \nabla \cdot \mathbf{U} & = 0, \\ \frac{\partial U_1}{\partial t} + \nabla \cdot (U_1 \mathbf{U}) & = -\frac{1}{\rho}\frac{\partial p}{\partial x_1} + \nabla \cdot (\nu_t \nabla U_1) - \frac{1}{\rho}\left[\frac{\partial(\rho U_1^2)}{\partial x_1} + \frac{\partial(\rho U_1 U_2)}{\partial x_2} + \frac{\partial(\rho U_1 U_3)}{\partial x_3}\right], \\ \frac{\partial U_2}{\partial t} + \nabla \cdot (U_2 \mathbf{U}) & = -\frac{1}{\rho}\frac{\partial p}{\partial x_2} + \nabla \cdot (\nu_t \nabla U_2) - \frac{1}{\rho}\left[\frac{\partial(\rho U_2 U_1)}{\partial x_1} + \frac{\partial(\rho U_2^2)}{\partial x_2} + \frac{\partial(\rho U_2 U_3)}{\partial x_3}\right], \\ \frac{\partial U_3}{\partial t} + \nabla \cdot (U_3 \mathbf{U}) & = -\frac{1}{\rho}\frac{\partial p}{\partial x_3} + \nabla \cdot (\nu_t \nabla U_3) - \frac{1}{\rho}\left[\frac{\partial(\rho U_1 U_3)}{\partial x_1} + \frac{\partial(\rho U_2 U_3)}{\partial x_2} + \frac{\partial(\rho U_{3)}^2}{\partial x_3}\right], \end{cases} \tag{2.12}$$

with the system supplemented by the prescribed conditions on the boundary patches given in Figure 2.1.

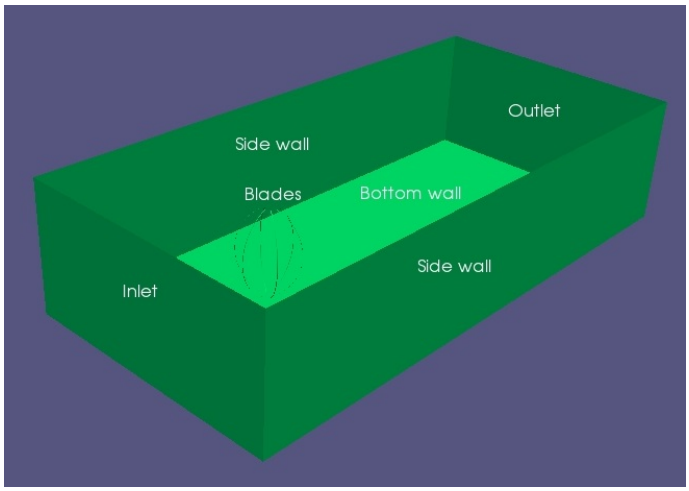


Figure 2.1: The Lux VAWT geometry and boundary patches

2.1.3 Initial and Boundary Conditions

The boundary conditions on the incompressible Navier–Stokes equations (2.12) can be expressed based on the aerodynamic and geometrical flow domain. The flow conditions are just as important as the differential equations that determine the flow equations, and the outcomes of the simulations rely on the inlet and outlet conditions and the conditions at the boundary walls of the computational domain. For example:

- At the entry (`Inlet`) of the domain depicted in Figure 2.1, the value of $\mathbf{U}$ and p are prescribed as

$$\mathbf{U} = \mathbf{U}_0, \ \frac{\partial p}{\partial n} = 0, \tag{2.13}$$

  where $\mathbf{U}_0$ is taken to be the constant inlet velocity ranging from 4 m/s – 22 m/s along the flow direction.

- At the side walls of the domain, a symmetry boundary condition is prescribed by assigning the flow fluxes or the normal gradients of all scalar variables to zero. The wall-normal velocity component is set to zero because no flow crosses a symmetry patch. Mathematically, this is written as

$$\frac{\partial \mathbf{U}}{\partial n} = 0, \ \frac{\partial p}{\partial n} = 0. \tag{2.14}$$

- On the turbine blade boundaries, a moving wall condition (for moving or rotating walls) is prescribed for the flow velocity, and the pressure values on the blade surface are then estimated, while the pressure gradients are fixed to zero. That is,

$$\mathbf{U} = \mathbf{U}_b, \ \frac{\partial p}{\partial n} = 0. \tag{2.15}$$

  where $\mathbf{U}_b$ is the velocity of the blade.

- At the bottom wall, a no-slip boundary condition is imposed on the velocity field to model the ground. That is, the velocity of the flow field at the bottom wall is set as wall velocity so that a fixed value

boundary condition applied. Given that the bottom wall is stationary, the fluid velocity is set to zero. For the pressure field, the zero-gradient condition is imposed,

$$\mathbf{U} = \mathbf{0},\ \frac{\partial p}{\partial n} = 0, \tag{2.16}$$

Outlet: At the exit plane, the convective outflow boundary condition [39] is imposed on the velocity and pressure fields. That is,

$$\frac{\partial \mathbf{U}}{\partial t} + \left(\mathbf{U}_c \times \frac{\partial \mathbf{U}}{\partial n}\right) = \mathbf{0},\ -p\mathbf{n} + \mu\frac{\partial \mathbf{U}}{\partial n} = \mathbf{0}, \tag{2.17}$$

where $\mathbf{U}_c$ is the advective velocity at the location of the outlet patch and $\mathbf{n}$ is a normal vector to the outlet boundary. However, when the flow is fully developed at the outlet boundary, one can set $\frac{\partial \mathbf{U}}{\partial n}$ to zero. Therefore,

$$\frac{\partial \mathbf{U}}{\partial t} = \mathbf{0},\ p\mathbf{n} = \mathbf{0}. \tag{2.18}$$

2.1.4 Effects of Reynolds number

The Reynolds number (Re) [107] is another parameter of interest in fluid dynamics used to characterize flow behaviors and predict flow patterns. This dimensionless parameter plays a notable role in predicting the patterns in a fluid and airfoil behavior. It is a ratio of the inertial force of the moving fluid to the viscous force acting on the fluid particles determining if the fluid flow is laminar or turbulent. Equation (1.1) suggests that Re quantitatively depends on the flow density, the velocity of the flow, the characteristic length scale of the flow surface, and the fluid kinematic viscosity. When Re decreases, the relative magnitude of the viscous forces dominates the entire flow domain and characterizes into an increased pressure gradient and drag coefficient [91].

Most flows experienced in engineering practice become unsettled at a given Reynolds number [83]. For example, a fully developed laminar flow in a pipe has a low Reynolds number (e.g., Re $\leq$ 2300), whereas a small swirl or a puff wind can be categorized as a turbulent flow with a moderately high Reynolds number. At high Reynolds number, the flow dynamics characteristically become more chaotic, non-linear, time-dependent, and turbulent [71]. As a result, the Navier–Stokes equations become notably difficult to solve, particularly at the presence of the turbulence eddies. The qualitative analysis of the solution to the Navier– Stokes equations reveals that turbulence evolves as instability of laminar flow [87].

2.2 Turbulence Modeling

Most fluid flows of engineering interest are turbulent in nature [19] and therefore may require special treatment to understand their viscous properties. Turbulence simply illustrates an irregular motion of fluid particles, especially around a solid surface in a flow path. This implies a case of a chaotic and unsteady random state of

motion in a flow domain [32]. At relatively high Reynolds numbers, an orderly laminar flow propagates into a turbulent flow with a radical change to the flow character, causing intermittent changes to the velocity and pressure of the flow [130]. Turbulence is identified by the randomness in the flow, increased diffusivity, and energy dissipation ([68], [120]). Turbulence is three dimensional, rotational, and unsteady with a large range of scale motions including the whirling flow structures, otherwise called turbulent eddies, which instigate continuous variation in velocity and pressure [85]. These eddies are large in scale, ranging from the least turbulence eddies characterized by Kolmogorov microscales to the size of the order of the flow geometry [58]. The ratio of the smallest eddies to the largest scales decreases rapidly as the Reynolds number increases and the kinematic energy of the largest scales dissipated over time through the presence and interaction with the smaller eddies. Therefore, it is imperative to resolve all physically relevant scales to accurately numerically simulate turbulent flow. Although the exact physical nature of turbulence and its characteristics have not been fully understood, it can only be modeled based on one's concept of what the viscous nonlinear transfer terms should be to estimate the effect of turbulence in the flow. There are several approaches to simulate turbulence, for example, Direct Numerical Simulation (DNS), Large Eddy Simulation (LES) or based on Reynolds-Averaged Navier–Stokes (RANS) methods. Within these approaches, there are multiple models, with various strengths and weaknesses.

2.2.1 Direct Numerical Simulation

Direct numerical simulation (DNS) of turbulent flow involves the computation of numerical solutions to the Navier–Stokes equations without adopting any additional approximation or turbulence model [52]. In this approach, the flow equations are integrated numerically over the whole range of the turbulent scales to estimate the mean flow and every turbulent velocity fluctuation. The unsteady Navier–Stokes equations are solved on a sufficiently dense computational grid to resolve all turbulence scales in space and time [130]. Based on these requirements and the demand for high computational power, it is practically inefficient to apply DNS to complex flows of industrial applications. Alternatively, fluid dynamics researchers have applied different approaches and employed various turbulence models to estimate the effects of turbulence on the flow. Common among these techniques are large eddy simulation (LES) and Reynolds-Averaged Navier–Stokes (RANS) simulation.

2.2.2 Large Eddy Simulation

Large eddy simulation (LES) is an intermediate form of turbulence simulation used to model and compute a certain range of eddy scales (large eddies) in a time-dependent simulation. This technique discards the smallest eddies rather than resolving the whole spectrum of turbulence scales within the flow domain. LES usually resolves the largest scale motions of turbulent flow and estimates or models the small-scale motions (small eddies) using a subgrid-scale (SGS) model [39]. The small eddies within the flow are generally observed to be nearly isotropic; hence the aerodynamic behavior does not undermine the physics of the mean flow [6].

The large eddies are basically anisotropic and actively interact with (and drain) energy from the mean flow.

LES employs the spatial averaging on the Navier–Stokes equations with a filtering function or a cutoff scale to separate the smaller eddies from the larger eddies [43]. After the separation, the aerodynamic behavior of the resulting smaller turbulent eddies is modeled via the subgrid-scale (SGS) model. The flow interaction effects between the large-scale resolved eddies and smaller-scale unresolved eddies result in the formation of the SGS stresses. Then, LES implements a statistical filtering procedure (spatial averaging) on the unsteady flow equations.

2.2.3 Reynolds-Averaged Navier–Stokes Simulation

Prior to solving Navier–Stokes equation (2.10)–(2.11) using this approach, the Unsteady Reynolds-Averaged Navier–Stokes (URANS) equations are obtained from the instantaneous Navier–Stokes equations by means of the Reynolds decomposition using time, spatial, and ensemble averaging techniques [58]. The Reynolds decomposition is used to represent any of the aforementioned averaging techniques, particularly when applying to the Navier–Stokes equations. It entails the process of separating the flow physical properties into mean (time-averaged) and fluctuating components [34]. The choice of the averaging techniques depends on the characteristics of the turbulent flow [51]. Given that most engineering problems involve inhomogeneous turbulence, the time averaging technique can be chosen as an appropriate form of the Reynolds decomposition. Hence, without any lost of generality, if we consider incompressible Newtonian fluids (such as air), the flow quantities $\mathbf{U}$ and p from equations (2.10)–(2.11) can be represented as the sum of a mean and fluctuating component [34] as follows

$$\mathbf{U}(\mathbf{x},t) = \overline{\mathbf{U}}(\mathbf{x},t) + \mathbf{U}^{'}(\mathbf{x},t), \tag{2.19}$$

$$p(\mathbf{x},t) = \overline{p}(\mathbf{x},t) + p'(\mathbf{x},t), \tag{2.20}$$

where

$$\overline{\mathbf{U}}(\mathbf{x},t) = \lim_{T\to\infty}\frac{1}{T}\int_{t}^{t+T}\mathbf{U}(\mathbf{x},t)\,\mathrm{d}t = \overline{\overline{\mathbf{U}}(\mathbf{x},t) + \mathbf{U}'(\mathbf{x},t)} = \overline{\mathbf{U}}(\mathbf{x},t) + \overline{\mathbf{U}'}(\mathbf{x},t). \tag{2.21}$$

Here, t represents the time and T is the averaging time interval that is chosen to capture the average transient scale of the flow fluctuation. The interesting case is the limit of $T \longrightarrow +\infty$. If T is sufficiently large, then $\overline{\mathbf{U}}(\mathbf{x},t)$ does not depend on the time at which the averaging is initiated. However, the long-time Reynolds average given by equation (2.21) is not sensitive to some flow features such as non-turbulent unsteadiness, which is characterized by transient or periodic behavior. In such instances, a better decomposition and averaging technique is required. Hence, we adopt ensemble averaging of the form

$$\overline{\mathbf{U}}(\mathbf{x},t) = \lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N}\mathbf{U}_i(\mathbf{x},t),$$

where N represents the number of identically performed experiments needed to mitigate fluctuation effects. For the URANS decomposition, the mean value is called an ensemble average or phase average, in contrast

to the time average in steady RANS. This averaging technique is suitable and applicable to any flow. Using equation (2.21), we deduce that $\mathbf{U}^{'}(\mathbf{x},t)=0$ and emphasize that averaging any linear term in the governing equations intuitively produces an equivalent term for the averaged quantity [34]. However, any nonlinear term often yields two terms: the product of the mean and covariance, for example:

$$\overline{p\mathbf{U}}=\overline{(\overline{p}(\mathbf{x},t)+p'(\mathbf{x},t))(\overline{\mathbf{U}}(\mathbf{x},t)+\mathbf{U}'(\mathbf{x},\mathbf{t}))}=\overline{p}(\mathbf{x},t)\overline{\mathbf{U}}(\mathbf{x},t)+\overline{p'(\mathbf{x},t)\mathbf{U}'(\mathbf{x},\mathbf{t})}. \tag{2.22}$$

The last term goes to zero if the two flow quantities are unrelated [51]. This is not always the case for turbulent flow, where the flow equations include terms such as turbulent scalar flux, which is often difficult to estimate uniquely in term of the average quantities.

With this basis, the URANS equations can be derived using equations (2.19)–(2.20) in the general Navier–Stokes equations by taking the Reynolds averaging to obtain

$$\nabla\cdot\overline{\mathbf{U}}=0, \tag{2.23}$$

$$\frac{\partial\overline{\mathbf{U}}}{\partial t}+\nabla\cdot(\overline{\mathbf{U}}\overline{\mathbf{U}})=\mathbf{g}-\frac{1}{\rho}\nabla\overline{p}+\nabla\cdot(\nu_t\nabla\overline{\mathbf{U}})-\nabla\cdot(\overline{\mathbf{U}'\mathbf{U}'}). \tag{2.24}$$

These equations can be represented in component form as

$$\begin{cases}\nabla\cdot\overline{\mathbf{U}} & =0,\\ \frac{\partial\overline{U_1}}{\partial t}+\nabla\cdot(\overline{U_1}\overline{\mathbf{U}}) & =-\frac{1}{\rho}\frac{\partial\overline{p}}{\partial x_1}+\nabla\cdot(\nu_t\nabla\overline{U_1})-\frac{1}{\rho}\left[\frac{\partial(\rho\overline{U_1'^2})}{\partial x_1}+\frac{\partial(\rho\overline{U_1'U_2'})}{\partial x_2}+\frac{\partial(\rho\overline{U_1'U_3'})}{\partial x_3}\right],\\ \frac{\partial\overline{U_2}}{\partial t}+\nabla\cdot(\overline{U_2}\overline{\mathbf{U}}) & =-\frac{1}{\rho}\frac{\partial\overline{p}}{\partial x_2}+\nabla\cdot(\nu_t\nabla\overline{U_2})-\frac{1}{\rho}\left[\frac{\partial(\rho\overline{U_2'U_1'})}{\partial x_1}+\frac{\partial(\rho\overline{U_2'^2})}{\partial x_2}+\frac{\partial(\rho\overline{U_2'U_3'})}{\partial x_3}\right],\\ \frac{\partial\overline{U_3}}{\partial t}+\nabla\cdot(\overline{U_3}\overline{\mathbf{U}}) & =-\frac{1}{\rho}\frac{\partial\overline{p}}{\partial x_3}+\nabla\cdot(\nu_t\nabla\overline{U_3})-\frac{1}{\rho}\left[\frac{\partial(\rho\overline{U_1'U_3'})}{\partial x_1}+\frac{\partial(\rho\overline{U_2'U_3'})}{\partial x_2}+\frac{\partial(\rho\overline{U_3'^2})}{\partial x_3}\right],\end{cases} \tag{2.25}$$

where the first equation (2.23) represents the continuity equation for the mean flow and the last three equations are the time-averaged $x_1,\ x_2,\ x_3$ -linear momentum equations. In equation (2.25), the left-hand side quantifies the momentum shift in a fluid element due to the fluctuation and convection in the mean flow and is balanced by the averaged pressure, viscous stress, and the additional source term ($\overline{\mathbf{U}'\mathbf{U}'}$) occasioned by the fluctuating velocity field, otherwise called the Reynolds stress. These Reynolds stresses characterize turbulent transport and stirring in the unsteady turbulent flows [34]. The extra turbulent stresses include the three normal stresses defined by

$$\tau_{x_ix_i}=-\rho\overline{U_i'^2}, \tag{2.26}$$

and the three shear stresses

$$\tau_{x_ix_j}=\tau_{x_jx_i}=-\rho\overline{U_i'U_j'},\quad i,\ j=1,2,3,\quad i\neq j. \tag{2.27}$$

The additional terms generated by the Reynolds decomposition of the Navier–Stokes equations raise the closure problem and require additional modeling [127]. For instance, equation (2.27) is for the symmetric stress

tensor with six independent components expressing the interaction between the fluctuating mean velocities. Although formulating higher-order correlation equations for the Reynolds stress tensor is attainable, they are likely to contain more and higher-order unknown correlations that may require extra modeling or approximations [130]. These additional approximations are numerically framed by turbulence modeling in CFD. To resolve turbulent flows with the URANS equations, it is essential to describe turbulent models to predict the Reynolds stress and close the URANS equations. Before we delve into the physics and the characteristic features of the turbulent models, the benefits of the Reynolds-Averaged Navier-Stokes simulation along with other simulation techniques in practice are briefly highlighted in Table 2.1.

CFD techniques	**Merits**	**Demerits**
RANS	• Excellent performance for many industrially relevant flows. • Modest computational cost.	• May not capture all the aerodynamics of the flow in great details. • Dependence on empirical correlations to start the simulation.
LES	• Capable of resolving the important unsteady flow structure. • Able to reproduce turbulence with much higher accuracy. • Relatively more accurate and applicable than the RANS. • Less computationally expensive and most viable numerical tool for simulating realistic transitional flows.	• Smallest eddies are detached, and modeled separately with a subgrid-scale (SGS) model. • Requires fine spatial and transient grid in the inertial subrange. • Requires a lot of space and memory of the computer resources.

DNS	• Appears to be the easiest and most accurate approach to simulate turbulent flows. • Does not require any empirical correlations for the simulations. • Does not use of any models and is equivalent to an ordinary laboratory experiment. • Incorporates distinct features to characterize all the flow details thereby useful in the advancement of turbulence models for practical flows. • Provides data for the evaluation of subgrid models for LES.	• Computationally intensive. • Remains extremely time-consuming for high Reynolds number flow.

Table 2.1: Comparison of the three computational techniques in CFD

2.3 Turbulence models

The purpose of the Reynolds-Averaged turbulent modeling is to describe the Reynolds stress tensor derived in equations (2.26)–(2.27) in terms of the known quantities. These Reynolds stress tensors can be computed following the Boussinesq approximation [18], which expresses the Reynolds stress tensor as a function of a mean rate of fluid and turbulent viscosity (ν_t). According to the report by Schmitt [112], Boussinesq proposed that the turbulence effect on flow can be quantified as a function of increased viscosity and suggested that the Reynolds stresses are proportional to mean rate of deformation (i.e., velocity gradients) in a linear form

$$\tau_{i,j} = -\rho\overline{U_i' U_j'} = \nu_t \left(\nabla \mathbf{U} + (\nabla \mathbf{U})^T\right) + \frac{2}{3} k \mathbf{I}, \tag{2.28}$$

where

$$k = \frac{1}{2}\overline{\mathbf{U}' \cdot \mathbf{U}'},$$

represents the turbulent kinetic energy and ν_t is the eddy viscosity. Equation (2.28) suggests that the turbulent momentum transport is directly related to the velocity gradient (i.e., the gradient of momentum per unit mass). Similarly, the turbulent transport of any given scalar quantity can be assumed to be proportional to the gradient of the mean value of the transported quantity. These turbulent models are called eddy viscosity models. The eddy viscosity ν_t can be estimated using different approaches starting from the algebraic relations with different local equilibrium assumptions to the solution of the transport equations. Based on the number of transport equations under consideration, the eddy viscosity model is categorized into many groups: 0-equation (algebraic), one-equation, two-equation, and Reynolds–stress models. 0-equation (algebraic) models are the easiest and simplest form of turbulence models with the turbulence length scale specified prior to the simulation and have limited applications in practical flows. They also fail to consider the previous history of the flow and assumes that turbulence is dissipated immediately on generation. One-equation and two-equation models are preferable for industrial applications [133] and incorporate a differential transport equation for turbulent velocity scales. In addition, two-equation models incorporate another transport equation for the length scale (or time scale) to predict eddy viscosity without the need for experimental data. The most popular family of the two-equation models is the k–ϵ and k–ω SST models, which are explained in the next subsections.

2.3.1 k–ϵ model

Fundamentally, when the eddy turbulent viscosity is modeled as a relation of turbulent kinematic energy, k, and turbulent dissipation energy, ϵ, we obtain a two-equation turbulence model, among which is the k–ϵ model, possibly the most common turbulence model in use today. The first version of k–ϵ was proposed by Jones and Launder [70] and has been improved upon with more parameters and features to estimate turbulence effect accurately. The most widely used among the family of k–ϵ models is the standard k–ϵ. Its eddy turbulent viscosity is generally expressed as

$$\nu_t = C_\mu \frac{k^2}{\epsilon}, \tag{2.29}$$

where C_μ is a dimensionless constant. To solve equation (2.29) for the eddy viscosity ν_t, two additional transport equations must be defined for k and ϵ within the flow domain. Based on the Boussinesq approximation reported in [70], the transport equations for k and ϵ are given as

$$\frac{\partial}{\partial t}(\rho k) + \nabla \cdot (\rho k \overline{\mathbf{U}}) = \nabla \cdot \left(\mu + \frac{\mu_t}{\sigma_{k2}} \nabla k\right) + P_k - \rho\epsilon, \tag{2.30}$$

$$\frac{\partial}{\partial t}(\rho \epsilon) + \nabla \cdot (\rho \epsilon \overline{\mathbf{U}}) = \nabla \cdot \left(\mu + \frac{\mu_t}{\sigma_{w2}} \nabla \epsilon\right) + C_{1\epsilon} \frac{\epsilon}{k} P_k - C_{2\epsilon} \rho \frac{\epsilon^2}{k}, \tag{2.31}$$

where the Boussinesq assumption occurs in the production term

$$P_k = 2\mu_t S_{ij} \cdot S_{ij},$$

with

$$S_{ij} = \frac{1}{2}\left(\frac{\partial \overline{\mathbf{U}}_i}{\partial x_j} + \frac{\partial \overline{\mathbf{U}}_j}{\partial x_i}\right),$$

to describe the generation of turbulent kinetic energy due to mean velocity gradients. The last term in equation (2.30) and (2.31) represents the rate of destruction of the k and ϵ, respectively. The k–ϵ model equations contain five controlling parameters, which can be adjusted to suit a different range of turbulent flows. For the standard k–ϵ model, the five constants (closure coefficients) are given in Table 2.2. Qualitatively, equations (2.31) suggests that a singularity can occur in the k–ϵ modeling especially near the wall where k likely vanished. To correct the anomaly during the computation, one can update the time scale $T_t = \frac{k}{\epsilon}$ to $T_t = \frac{k}{\epsilon} + T_k$, where $T_k = \left(\frac{\nu_t}{\epsilon}\right)^2$ is the Kolmogorov transient scale for the least turbulent eddies [134].

Standard k–ϵ Closure	**k–ω Closure**	**SST Closure**
$\sigma_{k2} = 1$	$\sigma_{k1} = 0.85$	
	$\beta_1 = 0.075$	$\beta^* = 0.09$
$\sigma_{\omega 2} = 1.30$	$\sigma_{\omega 1} = 0.65$	$a_1 = 0.31$
$C_\mu = 0.09$	$\gamma_1 = 0.553$	
$C_{1\epsilon} = 1.44$		
$C_{2\epsilon} = 1.92$		

Table 2.2: Closure coefficients for each turbulence model [93].

When describing a wall function, a flow condition must be set for the turbulent variables. For k and ϵ, the initial value can be calculated using

$$k = \frac{3}{2}\left(U_{inlet} T_u\right)^2, \; \epsilon = C_\mu^{\frac{3}{4}} \frac{k^{\frac{3}{2}}}{l},$$

where U_{inlet} represents the mean flow velocity at the inlet boundary, T_u the turbulence intensity, and l is the characteristic length scale of the flow.

2.3.2 k–ω model

For most two-equation models such as the standard k–ϵ model, the turbulent viscosity, ν_t, is described mathematically by the product of a flow velocity scale $\vartheta = \sqrt{k}$ and a length scale $\ell = \frac{k^{\frac{3}{2}}}{\epsilon}$. However, the turbulent kinetic energy dissipation rate, ϵ, is not the only determining variable to express the possible length scale of the flow. One distinct way of describing the turbulent length scale is via the k–ω model. The standard k–ω is a two-equation model, one for k and the other for ω. It was formulated by Wilcox [133] and employs the turbulent frequency $\omega = \frac{\epsilon}{k}$ so that the length scale of the model becomes

$$\ell = \frac{\sqrt{k}}{\omega}.$$

Using this length scale, the turbulent viscosity of the model is given as

$$\nu_t = \frac{k}{\omega}.$$

The Reynolds stresses are estimated by the Boussinesq approximation to obtain

$$\tau_{x_i x_j} = -\rho\overline{U_i' U_j'} = 2\mu_t S_{ij} - \frac{2}{3}\rho k\delta_{ij} = \rho_t\left(\frac{\partial\overline{U}_i}{\partial x_j} + \frac{\partial\overline{U}_j}{\partial x_i}\right) - \frac{2}{3}\rho k\delta_{ij}. \tag{2.32}$$

The transport equation for k and ω in the standard k–ω model are given as

$$\frac{\partial}{\partial t}(\rho k) + \nabla\cdot(\rho k\overline{\mathbf{U}}) = \nabla\cdot\left[\left(\mu + \frac{\mu_t}{\sigma_{k1}}\right)\nabla k\right] + P_k - \beta_*\rho k\omega, \tag{2.33}$$

$$\frac{\partial}{\partial t}(\rho\omega) + \nabla\cdot(\rho\omega\overline{\mathbf{U}}) = \nabla\cdot\left[\left(\mu + \frac{\mu_t}{\sigma_{\omega 1}}\right)\nabla\omega\right] + P_w - \beta_1\rho\omega^2, \tag{2.34}$$

where

$$P_k = 2\mu_t S_{ij}\cdot S_{ij} - \frac{2}{3}\rho k\frac{\partial\overline{U_i}}{\partial x_j}\delta_{ij} \;\; and \;\; P_w = \gamma_1\left(2\rho S_{ij}\cdot S_{ij} - \frac{2}{3}\rho\omega\frac{\partial\overline{U_i}}{\partial x_j}\delta_{ij}\right)$$

represent the production terms for the turbulent kinetic energy k and ω respectively. The last term depicts the rate of dissipation of the turbulent eddies. For practical application, the model is controlled by five regulating constants defined in Table 2.2. The k–ω model is simple and easier to use without fear of singularity especially for low Reynolds number flow applications. Wall-damping functions are not required near the wall region to estimate the turbulent effect in the boundary sublayer of the flow. Away from the boundary sublayer, the model tends to be influenced by the free-stream values of ω in the main flow [39].

2.3.3 k–ω SST model

In 1994, Menter [93] modified the standard k–ω with shear stress transport characteristics, calling the resulting turbulent model k–ω SST. The extension SST (shear stress transport) represents a formulation combining the best characteristics behavior of k–ω within the region of the boundary sublayer and the k–ϵ in the free shear flow. This is to circumvent the prevailing k–ω model problem of high sensitivity to the inlet free-stream turbulence properties. The k–ω SST model can be applied with no extra wall damping functions and as a low-Reynolds turbulence model for most industrial flows. The model improves the flow structures in the case of adverse pressure gradients (i.e., when the static pressure increases in the direction of the flow) and flow separation. The k–ω SST model includes a damped cross-diffusion derivative term in the ω ($= \frac{\epsilon}{k}$) transport equation, and by defining the turbulent eddy viscosity,

$$\nu_t = \frac{a_1 k}{\max(a_1\omega, SF_2)}, \tag{2.35}$$

to provide insight into the transport of the turbulent shear stress. The modified transport equations for k and ω in the k–ω SST model are

$$\frac{\partial}{\partial t}(\rho k) + \frac{\partial}{\partial x_j}\left(\rho\overline{U}_j k\right) = \rho P - \beta^*\rho\omega k + \frac{\partial}{\partial x_j}\left(\mu + \sigma_{k1}\mu_t\frac{\partial k}{\partial x_j}\right), \tag{2.36}$$

$$\frac{\partial}{\partial t}(\rho\omega) + \frac{\partial}{\partial x_j}\left(\rho\overline{U}_j\omega\right) = \frac{\gamma}{\nu_t}P_{wo} - \beta_1\rho\omega^2 + \frac{\partial}{\partial x_j}\left(\mu + \sigma_{\omega 1}\mu_t\frac{\partial\omega}{\partial x_j}\right) + 2(1-F_1)\frac{\rho\sigma_{\omega 1}}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial\omega}{\partial x_j}, \tag{2.37}$$

with the following variable definitions

$$P_{wo} = \tau_{ij}\frac{\partial \overline{U}_i}{\partial x_j},\ P = \min\left(P_{wo}, c_1\omega\right), \tag{2.38}$$

$$\overline{S} = \sqrt{2S_{ij}S_{j,i}}, \tag{2.39}$$

$$\tau_{ij} = \mu_t\left(2S_{ij} - \frac{2}{3}\frac{\partial \overline{U}_k}{\partial x_k}\delta_{ij}\right) - \frac{2}{3}\rho k\delta_{ij}, \tag{2.40}$$

$$\Phi = F_1\Phi_1 + (1 - F_1)\Phi_2, \tag{2.41}$$

$$F_1 = \tanh(\varphi_1^4), \tag{2.42}$$

$$F_2 = \tanh(\varphi_2^2), \tag{2.43}$$

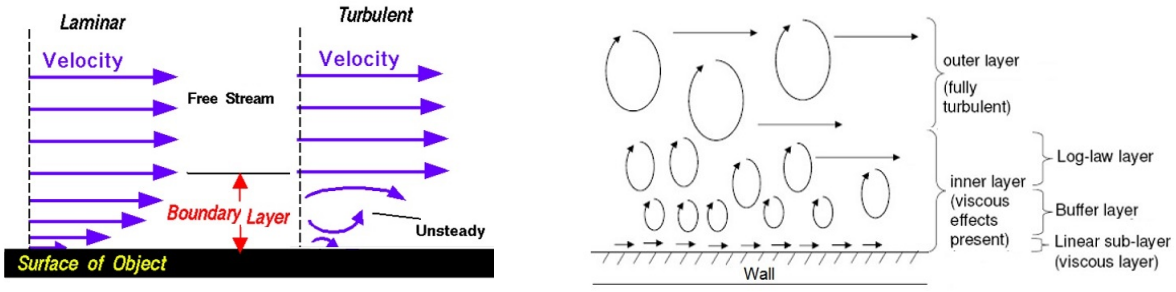
$$\varphi_1 = \min\left[\max\left(\frac{\sqrt{k}}{\beta^*\omega d}, \frac{500\nu}{d^2\omega}\right), \frac{4\rho\sigma_{\omega 2}k}{CD_{k\omega}d^2}\right], \tag{2.44}$$

$$\varphi_2 = \max\left(2\frac{\sqrt{k}}{\beta^*\omega d}, \frac{500\nu}{d^2\omega}\right),\ CD_{k\omega} = \max\left(2\rho\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j}, 10^{-20}\right), \tag{2.45}$$

and closure coefficients given in Table 2.2.

2.4 Boundary Layer

One of the primary challenges in CFD is how to characterize and estimate the turbulent viscosity in the vicinity of the boundary layer, where the viscous effects become significant. A boundary layer can be described as a thin layer of fluid in the immediate vicinity of a bounding surface where flow properties change stochastically due to the viscous force effects [118]. Fundamentally, the boundary layer can either be laminar or turbulent (see Figure 2.2a).



(a) Boundary layer

(b) Turbulent boundary layer domain

Figure 2.2: Boundary layer in the flow regime (From https://www.grc.nasa.gov)

In laminar flow, the boundary layer flow over the airfoil surface starts as a smooth flow (with no viscous effects) and increases in thickness as the flow continues towards the leading edge of the airfoil [111]. However, at the edge of the laminar flow is a transition region, where the smooth laminar flow breaks down as a result of the changes in the transport property of the flow. At this point, the flow decelerates swiftly, and the turbulent eddies or swirls are generated.

After the transition point, and when the flow is wholly developed, the viscous effects propagate and dominate the inertial effects, and a thin boundary viscous layer called the linear sub-layer is formed (see Figure 2.2b) [99]. In this region, the perceived velocity is usually low, and consequently the initial force is very small; however, the velocity gradient is large, assisting viscous force effects on the flow. Far away from the linear sub-layer is the region called log-law layer. Here, the inertial effects are dominant over the viscous effects. That is, the turbulent stresses are characteristically large while the viscous stress is gradually dissipated [131]. The intermediate boundary layer between the linear sub-layer and the log-law layer is called the buffer layer where the viscous shear stresses are compensated for by the turbulent shear stresses and are about equally significant.

2.4.1 Near Wall Modeling

The mathematical modeling of the area in the immediate vicinity of the wall is essential for accurate predictions of the wall-bounded turbulent flow field. Within the immediate vicinity of the solid wall in a flow, the Reynolds number is normally small, and the viscous effects are rapid and significant [20]. The thickness of the boundary sub-layer, however, decreases when the local Reynolds number increases. This near-wall, viscosity-dominated layer of turbulent flow can be estimated by including the near-wall damping and other source terms in the modeled transport equations. However, the most reliable modeling approach is to use a fine grid and low Reynolds number turbulence models. Low Reynolds number turbulence models can be used to simulate the damping effects at the wall; however, this must be applied on sufficiently dense computational grids [48]. Because of the high normal gradients of the velocity (and turbulence energies) across the viscous layer, fine grids are required to resolve the viscous layer flow and to provide adequate numerical resolution of the flow field.

An alternative approach to resolving the viscous boundary layer flow is to use the wall functions along with the coarse computational grids near the wall [73]. The approach does not entirely resolve the viscous sub-layer but applies the wall functions on the near-wall region treated by the coarse grids. The theoretical characterization of the flow profiles between the boundary layer surface and the first near-wall grid are approximated and superimposed [68]. The idea is to localize the first computational grid at the inception of the viscous sub-layer and appropriate the behavior of the velocity profile near the wall in order to predict the wall shear stress. For example, high Reynolds number turbulence models (such as the standard k–ϵ model) can resolve the near-wall layer through the use of wall functions. The wall functions apply the local equilibrium assumption and predict the production of k and the corresponding value of the turbulent

dissipation ϵ in the near-wall grid.

2.4.2 Wall Functions

The wall functions are a set of semi-empirical formulas and functions that describe the behavior of the flow in the logarithm sub-layer and assuming that the turbulence close to the near-wall region is a function only of the flow conditions at the boundary wall. The law of the wall [113], which describes the turbulent velocity profile near the wall, emphasizes that the mean velocity of a turbulent shear flow over a solid surface at a specific point within the boundary domain is a function of the logarithm of the distance from the boundary of the fluid domain. This near-wall area of steep gradients is modeled by the wall functions, which constitute an extra term in the momentum equation in order to account for the increased shear stress at the wall. Wall functions are generally applied to connect the inner field between the wall and the turbulence fully developed region. The use of wall functions, however, depends on the y^+, the dimensionless distance of the boundary wall (see Figure 2.3) and is calculated by

$$y^+ = \frac{y\overline{u}^*}{\nu_t}, \tag{2.46}$$

where $\overline{u}^* = \sqrt{\frac{\tau_w}{\rho}}$ represents the friction velocity, ν_t is the kinematic viscosity, and y and τ_w is the absolute distance from the wall and wall shear stress respectively. The y^+ value can be interpreted relative to a local Reynolds number, whose magnitude can be used to quantify the relative importance of viscous and turbulence effects.

Empirical observations and simulations suggest that turbulence effect is small, and perhaps negligible, in the viscous sub-layer (linear sub-layer) where the first internal grid point is located within $0 < y^+ < 5$ [99].

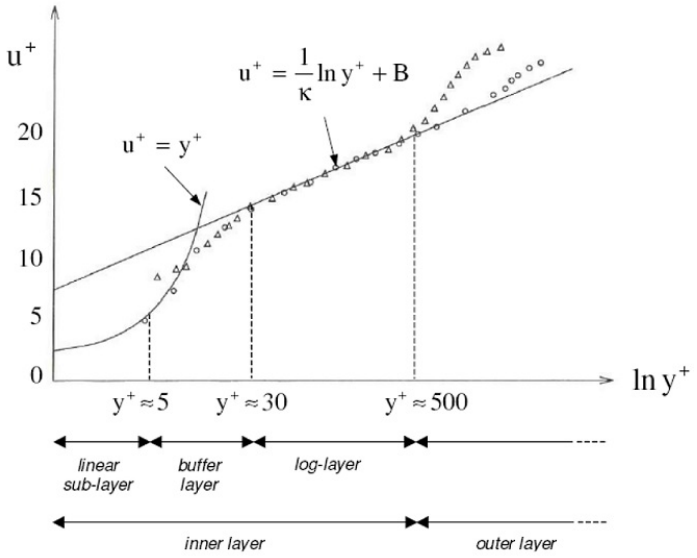


Figure 2.3: The normalized plot of the velocity profile near the wall [113]

Consequently, no wall functions are required to resolve an already resolved area because the mesh has a fine resolution at the wall. However, at the buffer layer ($5 < y^+ < 30$) and log layer ($30 < y^+ < 500$), the viscous effects are significant and require appropriate wall functions to resolve the boundary layer flow. In this research, standard wall functions have been applied to model the physics of the boundary layer flow near-wall region of the turbine blade. Standard wall functions are constructed based on the proposal of Launder and Spalding [84], and work reasonably well for a majority of high Reynolds number wall-bounded turbulent flows.

2.5 Approaches to Fluid Flow Problems

Analyzing fluid flow systems in engineering and environmental physical processes, including turbine aerodynamic flows, is a demanding task that requires both knowledge of the problem and the available resources. There have been many systematic methodologies designed to investigate various fluid dynamical problems that arise in nature. Fundamental among these approaches are experimental techniques, analytical fluid dynamics (AFD) techniques, and CFD techniques.

The experimental approach focuses on designing an experiment to investigate fluid flow problems of practical value. For instance, wind tunnel tests are a typical example of an experimental fluid dynamic approach to investigate and analyze the aerodynamic performance of wind turbines system. It is an effective approach when equipped with the necessary resources.

Instead of experimental techniques, the AFD approach generally requires considerable simplification of the governing equations of fluid motion and analytically predicts an approximate solution to the problem. It is mostly applied to problems with simple geometry and physics.

The CFD approach, however, deals with the methodology for obtaining discrete solutions from fluid dynamical problems. It entails the process of simulating a fluid dynamical system using modeling and numerical techniques.

2.5.1 Domain Discretization Techniques

Discretization is a significant step in any meaningful numerical simulation adopting computational fluid dynamics entailing a process of converting continuous functions (physical quantities like velocity, temperature, chemical concentration, etc.), numerical models, and or governing equations into a discrete form that can be handled by computer simulation.

The discretization of the problem domain generates a mesh system on which the transport equations are solved [76]. The resulting mesh system enables the transformation of the flow equations to a corresponding system of algebraic equations, at the same time reducing the number of values a continuous transport quantity assumes by grouping them into finite intervals or bins. The solution to the algebraic system culminates in a set of finite values that correspond to the solution of the original flow equations at some set positions in

space and time, as long as the flow conditions are satisfied. The problem domain is partitioned into a finite number of discrete regions called control volumes or cells illustrated in Figure 2.4, and for unsteady flow, the time domain is further split into a discrete number of time-steps. Methods such as the finite volume, finite difference, and finite element discretize the control volumes or cells of the problem following:

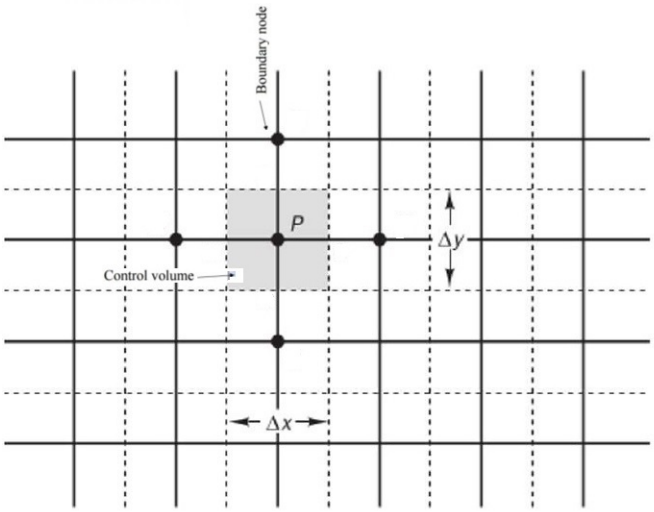


Figure 2.4: Computational grid showing control volume and nodes.

- Spatial discretization, which involves discretization of the problem domain along a spatial variable using a discrete set of points to bound a region of the domain when connected.
- Temporal discretization, which discretizes the time variable of the problem domain into a finite number of time intervals or discrete steps.

2.5.1.1 Finite Difference Discretization

A finite difference discretization is a typical numerical discretization method of finding an approximate solution to a differential equation using an approximate derivative based on discrete values positioned at spatial and temporal points. To solve equations (2.25), we can replace the derivative terms of the equations with finite approximations using Taylor series [80], which provide a systematic way of formulating an approximation to higher-order derivatives of any order as long as the unknown physical quantity is sufficiently smooth. Sequential application of the finite difference discretization at internal grid nodes generates a system of linear or nonlinear algebraic equations. These algebraic equations can be solved by the iterative solvers to find the values of the dependent variables $(\overline{p}, \overline{U})$ at each grid points in the discretized domain. The challenges in finite difference discretization applications lie in the inherent errors of approximation of the derivative terms and the difficulty of implementation for complex geometry. However, when the grid spacing and time step

are chosen sufficiently small and the resulting algebraic equations solved accordingly, an error due to finite differencing can be small. The finite difference discretization is generally applicable to a structured grid of a simple domain [123].

2.5.1.2 Finite Volume Discretization

A finite volume (FV) discretization combines both the characteristic attributes of the finite difference method and finite element method to resolve flow problems computationally. A finite volume is a discretization technique for PDEs, particularly those that arise from physical conservation laws, based on the control volume formulation [77]. FV is one of the applicable discretization methods for many industrial CFD codes, such as OpenFOAM and Fluent, to solve fluid flow equations in complex geometries. It facilitates the conservation of the transport quantities especially across shocks and other discontinuities [132].
A finite volume discretization is based on solving the transport equations within a defined control volume. The governing transport equations are integrated over the control volume, and the resulting equations are then discretized to generate a system of algebraic equations. These equations can then be solved with numerical iterative methods from specified initial conditions. For a given aerodynamic flow, the adopted FV depends on the type of control volume and the integral evaluation techniques. The control volumes have different shapes (e.g., tetrahedrons, hexahedrons, prisms, pyramids, dodecahedron, and so on). Fundamentally, a typical control volume can be of any cell shape provided that the fluxes entering the volume are equivalent to those leaving the neighboring volume and preserve the flow of physical quantity. However, the conservative property of the flow may be affected by numerical diffusion, especially when all the fluxes move through a given cell at a single time step. In the thesis, we have adopted the finite volume method on a flexible hexahedral control volume.

2.5.2 Characteristic features of discretization schemes in CFD

In practice, numerical computations are done on a finite number of cells and the resulting computational data can only be physically realistic if the discretization schemes satisfy basic properties. These fundamental properties include:

- Consistency: Consistency is perhaps the crucial property of any numerical discretization methods. It deals with the extent to which the difference equations (obtained from a given discretization scheme) approximate the governing equations of the system. A consistent discretization scheme produces a numerical result that is asymptotically accurate [80]. For a discretization scheme to be consistent, the numerical errors such as truncation error terms and round-off error must tend to zero when the grid spacing $\Delta t \longrightarrow 0$ or $\Delta x \longrightarrow 0$. In this case, the exact solution to the governing equations satisfies the algebraic equations from discretization, at least up to the first-order of the discretization parameters.

- Stability: Stability simply describes the quantitative measure of the well-posedness of the flow problem

[30]. A stable numerical approximation means that the norm of the numerical errors (truncation, round-off, etc.) at any stage of the computation remains bounded as the computation continues from one marching step to the next [39]. Stability of a numerical scheme can be analyzed following von Neumann's method [79]. That is, the stability for finite-difference approximations of time-dependent problems can be investigated by analyzing the growth rate of the initial condition in terms of a wave (von Neumann stability analysis). More details on the stability of discretization schemes and its related concepts are given in section 2.4.5.

- Convergence: A convergent scheme implies that the numerical solution of the discretized equations tends towards the exact solution of the differential equation as the grid spacing tends to zero [39]. In numerical computation, solution convergence can be verified by running the same simulation on a sequence of refined grids with various step sizes.

- Conservation: The underlying conservation laws for fluid transport properties should be respected at all discrete levels of the computation. The integration of the flow governing equations over a finite control volumes generates a set of discretized conservation equations with fluxes of the transported property through the control volume faces. To ensure conservation of the fluid transport property for the whole computational domain, the flux of the transport property escaping a control volume through the face must be identical to the flux moving into the adjacent control volumes through the same face in a consistent manner. In addition, any artificial sources or sinks should be avoided by the discretization scheme to accurately resolve the flow.

- Boundedness: Numerical values generated by the discretization scheme must be within physical limits. For example, for a linear problem with no source, the bounds can be the maximum and minimum boundary values. However, fluid flow problems are commonly nonlinear and the transport properties of the flow may be outside the limit of boundary values. For instance, the value of the flow density and temperature must be nonnegative and free of spurious wiggles, and volume and mass fractions should be bounded by 0 and 1.

- Transportiveness: Transportiveness characterizes the influence of upstream nodes on the downstream nodes of the numerical scheme when solving convective flows. For instance, in most fluid flow problems, diffusion usually propagates in all directions but convection only in the flow direction. A given discretization scheme should be sensitive to the flow direction and resolve it accordingly.

2.5.3 Discretization of the Transport Equation

The general form of the unsteady flow governing equation for any scalar transport property ϕ can be expressed as

$$\underbrace{\frac{\partial}{\partial t}(\rho\phi)}_{\text{temporal derivative}} + \underbrace{\nabla\cdot(\rho\mathbf{U}\phi)}_{\text{convection term}} - \underbrace{\nabla\cdot(\rho\Gamma_{\phi}\nabla\phi)}_{\text{diffusion term}} = \underbrace{Q^{\phi}}_{\text{source term}}. \tag{2.47}$$

where Γ_ϕ is the diffusion coefficient and Q^ϕ is the volume source of ϕ. The stepwise discretization of equation (2.47) is described in two ways: discretization of space and time derivative terms.

2.5.4 Spatial Discretization

In this research, the spatial discretization technique is described to reflect the finite volume discretization extensively applied in the OpenFOAM codes and libraries. The discretization of space by the finite volume method entails the partitioning of the problem domain into a series of control volumes [68].

Figure 2.5 illustrates a regular example of a single control volume out of many control volumes in the domain of interest. The computational point P is situated at the centroid of the control volume to evaluate the variable of interest.

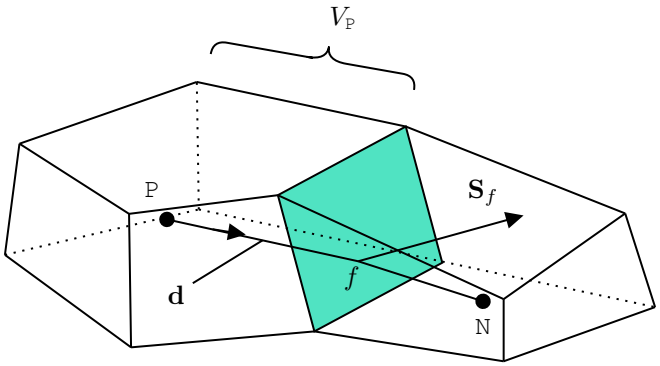


Figure 2.5: Control volume in the problem domain.

The polyhedron shape control volume is constrained by a set of boundary faces, and each face is shared with only one adjacent control volume. Basically, there are two groups of cell faces: internal faces, which lies between any two consecutive control volumes, and boundary faces, which align with the boundaries of the problem domain [68]. The computational points at the owner and neighbor control volumes are labeled P and N respectively. The shared face between the two identified control volumes is denoted by f at the centroid of the face. V_P denotes the finite volume of the representative control volume at the centroid P. $\mathbf{S}_f$ is constructed to represent the face area vector of any internal face, pointing towards the neighboring control volumes within the problem domain and a vector $\mathbf{d}$ connecting the centers of any two control volumes along the flow direction.

The face center and cell volume of the variable of interest can be estimated by applying a decomposition into triangles or pyramids. However, the positions of the computational points P and f at the centroid of the control volume and shared face can be defined by $\mathbf{x}_\mathrm{P}$ and $\mathbf{x}_f$, such that

$$\int_{V_\mathrm{P}} (\mathbf{x} - \mathbf{x}_\mathrm{P})\, \mathrm{d}V = 0,$$

$$\int_{f} (\mathbf{x} - \mathbf{x}_\mathrm{P})\, \mathrm{d}\mathbf{S} = 0.$$

By the finite volume approach, all the unknown transport properties share the same control volume. The transport equation (2.47) is required to be satisfied over each control volume $V_{\mathtt{P}}$ in the integral form [68]

$$\int_t^{t+\Delta t}\left[\frac{\partial}{\partial t}\int_{V_{\mathtt{P}}}(\rho\phi)\,\mathrm{d}V+\int_{V_{\mathtt{P}}}\nabla\cdot(\rho\mathbf{U}\phi)\,\mathrm{d}V-\int_{V_{\mathtt{P}}}\nabla\cdot(\rho\Gamma_\phi\nabla\phi)\,\mathrm{d}V\right]\mathrm{d}t=\int_t^{t+\Delta t}\left[\int_{V_{\mathtt{P}}}Q^\phi\,\mathrm{d}V\right]\mathrm{d}t. \tag{2.48}$$

In the steps that follow, equation (2.48) is discretized term by term by applying the generalized form of the identities

$$\int_V(\nabla\cdot\mathbf{A})\,\mathrm{d}V=\oint_{\partial V}\mathbf{A}\cdot\mathrm{d}\mathbf{S},$$
$$\int_V\nabla\phi\,\mathrm{d}V=\oint_{\partial V}\phi\;\mathrm{d}\mathbf{S},$$
$$\int_V\nabla\mathbf{A}\,\mathrm{d}V=\oint_{\partial V}\mathbf{A}\;\mathrm{d}\mathbf{S},$$

to convert the quantity in volume integral to the surface integral. In the above equations, ∂V denotes the closed surface bounding volume V, $\mathbf{A}$ is an arbitrary vector function, and $d\mathbf{S}$ characterizes the negligible surface element corresponding to the outward pointing normal on ∂V. The accuracy of the discretization method relies on the presumed fluctuation of the scalar function $\phi=\phi(\mathbf{x},t)$ in space and time about the point $\mathtt{P}$. Following the order of the equation (2.47), we require second-order accuracy for the adopted discretization scheme. To achieve this, the variation of the scalar function ϕ is assumed to be linear in both space and time so that

$$\phi(\mathbf{x})=\phi_{\mathtt{P}}+(\mathbf{x}-\mathbf{x}_{\mathtt{P}})\cdot(\nabla\phi)_{\mathtt{P}}, \tag{2.49}$$
$$\phi(t+\Delta t)=\phi^t+\Delta t\left(\frac{\partial\phi}{\partial t}\right),$$

with notation that

$$\phi_{\mathtt{P}}=\phi(\mathbf{x}_{\mathtt{P}}),$$

$$\phi^t=\phi(t).$$

Taking into consideration the proposed variation of ϕ over the control volume $V_{\mathtt{P}}$, equation (2.49), it implies

$$\int_{V_{\mathtt{P}}}\phi(\mathbf{x})\,\mathrm{d}V=\int_{V_{\mathtt{P}}}(\phi_{\mathtt{P}}+(\mathbf{x}-\mathbf{x}_{\mathtt{P}})\cdot(\nabla\phi)_{\mathtt{P}})\,\mathrm{d}V, \tag{2.50}$$
$$=\phi_{\mathtt{P}}\int_{V_{\mathtt{P}}}\mathrm{d}V+\left[\int_{V_{\mathtt{P}}}(\mathbf{x}-\mathbf{x}_{\mathtt{P}})\,\mathrm{d}V\right]\cdot(\nabla\phi)_{\mathtt{P}}, \tag{2.51}$$
$$=\phi_{\mathtt{P}}V_{\mathtt{P}}. \tag{2.52}$$

The second integral of equation (2.50) goes to zero at the centroid of the control volume. Following the same procedure, the integral of the divergence and gradient terms can be converted into a sum of integrals over all faces in the control volumes:

$$\int_{V_{\mathtt{P}}}\nabla\cdot(\mathbf{U}\phi)\,\mathrm{d}V=\sum_f\mathbf{S}_f\cdot(\mathbf{U}\phi)_f, \tag{2.53}$$

$$\int_{V_{\mathtt{P}}} \nabla\phi \,\mathrm{d}V = \sum_f \mathbf{S}_f \phi_f, \tag{2.54}$$

Convection term

By applying equations (2.50) through (2.54) on equation (2.48), the discretization of the convection term gives

$$\int_{V_{\mathtt{P}}} \nabla \cdot (\rho\mathbf{U}\phi) \,\mathrm{d}V = \sum_f \mathbf{S}_f \cdot (\rho\mathbf{U}\phi)_f, \tag{2.55}$$

$$= \sum_f \mathbf{S}_f \cdot (\rho\mathbf{U})_f \phi_f, \tag{2.56}$$

$$= \sum_f F\phi_f, \tag{2.57}$$

where F denotes the mass flux through the neighbor face. In order to quantify the convection term explicitly, equation (2.55) needs the face value of the scalar property ϕ. Variable values are located at the center of the control volume and in order to resolve the governing equations for the transport quantity, the values at the face centers are required in several places. This value can be calculated from the values of the cell centers using any differencing (interpolation) schemes. The next section describes some of the interpolation schemes as implemented in the open source package–OpenFOAM.

2.5.4.1 Central Differencing Scheme (CDS)

In this case, the face value of the transport quantity ϕ is estimated by the linear interpolation with the values in the two nearest adjacent nodes $\mathtt{P}$ and $\mathtt{N}$ of the control volume as shown in Figure 2.6

$$\phi_f \approx \lambda_f \phi_{\mathtt{N}} + (1 - \lambda_f)\phi_{\mathtt{P}}.$$

The interpolation factor λ_f is defined as the ratio of distances $|f\mathtt{P}|$ and $|\mathtt{NP}|$

$$\lambda_f = \frac{|f\mathtt{P}|}{|\mathtt{NP}|} = \frac{x_f - x_{\mathtt{P}}}{x_{\mathtt{N}} - x_{\mathtt{P}}}.$$

The scheme has second-order interpolation error [39] as can be seen from the Taylor series expansion of ϕ around the point x_f.

$$\phi(x) = \phi_{\mathtt{P}} + (x - x_{\mathtt{P}}) \left(\frac{\partial\phi}{\partial x}\right)_{\mathtt{P}} + \frac{(x - x_{\mathtt{P}})^2}{2} \left(\frac{\partial^2\phi}{\partial x^2}\right)_{\mathtt{P}} + \cdots \tag{2.58}$$

The evaluation of equation (2.58) at the nodal points x_f and $x_{\mathtt{N}}$ and their difference lead to

$$\phi_f = \lambda_f \phi_{\mathtt{N}} + (1 - \lambda_f)\phi_{\mathtt{P}} - \frac{(x_f - x_{\mathtt{P}})(x_{\mathtt{N}} - x_f)}{2} \left(\frac{\partial^2\phi}{\partial x^2}\right)_{\mathtt{P}} + \cdots \tag{2.59}$$

only under suitable assumptions. The CDS can cause numerical fluctuations in the approximation of the transport fluxes [59]. To regulate these possible oscillations, the upwind interpolation scheme can be applied to accurately estimate the face value of the transport quantity.

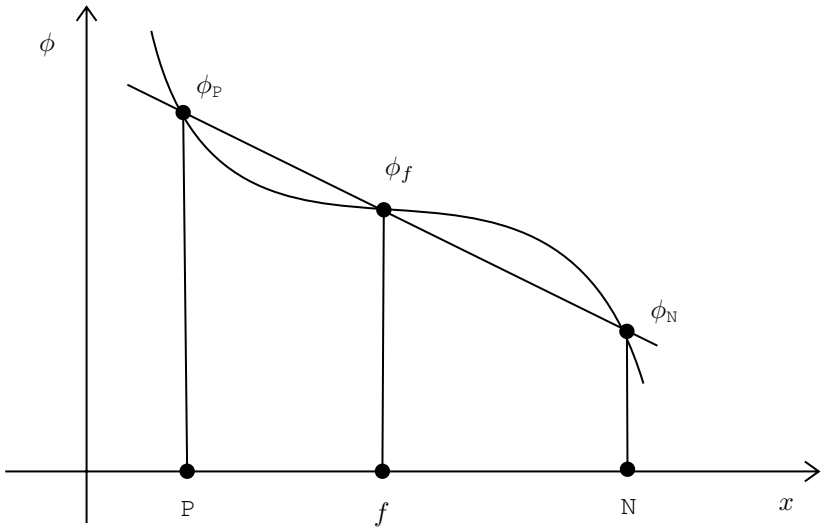


Figure 2.6: Face interpolation of ϕ with CDS method.

2.5.4.2 Upwind Interpolation Scheme

The upwind interpolation scheme, otherwise known as the upwind differencing scheme (UDS), is obtained when the convected value of ϕ at a face is approximated by its values at an upstream cell [110]. That is, the scalar property ϕ_f is quantitatively resolved according to the direction of the flow as follows

$$\phi_f = \begin{cases} \phi_f = \phi_{\mathrm{P}}, & \text{if a mass flux moves out of cell P into N ,} \\ \phi_f = \phi_{\mathrm{N}}, & \text{if a mass flux moves out of cell N into P.} \end{cases} \tag{2.60}$$

The computational benefit of the UDS is evident in meeting boundedness (i.e., convergence) criterion with oscillatory-free solutions. However, UDS often propagates numerical diffusion that can undermine the accuracy of the solution. This can be illustrated via a Taylor series expansion of ϕ around the point x_{P}, and evaluate the resulting expression at the point x_f, to obtain

$$\phi_f = \phi_{\mathrm{P}} + (x_f - x_{\mathrm{P}} \left(\frac{\partial \phi}{\partial x}\right)_{\mathrm{P}} + \frac{(x_{\mathrm{P}} - x_f)^2}{2} \left(\frac{\partial^2 \phi}{\partial x^2}\right)_{\mathrm{P}} + \cdots.$$

The evaluation implies that the UDS can be characterized as a first-order interpolation scheme. However, the dominant error term can distort the accuracy of the convective flux [68].

2.5.4.3 Linear Upwind Differencing Scheme

The linear upwind differencing scheme (LUDS) combines the best characteristic attributes of both the central and upwind differencing schemes to approximate the face value of ϕ. The scheme is proposed as a second-order correction to the first-order upwind differencing scheme to improve numerical stability [110]. The face value of the transport variable ϕ is estimated by

$$\phi_f = \phi + \nabla\phi \cdot \mathbf{k}, \tag{2.61}$$

where $\nabla\phi$ and ϕ represent the gradient- and cell-centered values of the transport variable respectively at the upstream cell, and $\mathbf{k}$ represents the displacement vector from the geometric center of the upstream cell to the face centroid. Equation (2.61) implies that the value of $\nabla\phi$ must be evaluated in each cell of the control volume. By approximating the gradient value at the cell P and N (Figure 2.6), equation (2.61) can be written as

$$\phi_f = \begin{cases} \phi_f = \phi_{\mathtt{P}} + (\nabla\phi)_{\mathtt{P}} \cdot \mathbf{k}_{\mathtt{P}f} & \text{if a mass flux moves out of cell } \mathtt{P} \text{ into } \mathtt{N}, \\ \phi_f = \phi_{\mathtt{N}} + (\nabla\phi)_{\mathtt{N}} \cdot \mathbf{k}_{\mathtt{N}f} & \text{if a mass flux moves out of cell } \mathtt{N} \text{ into } \mathtt{P}. \end{cases} \tag{2.62}$$

2.5.4.4 Linear Upwind Stabilized Transport

The linear upwind stabilized transport (LUST) is also known as the blended differencing scheme [68]. It is designed in such a way that the linear upwind scheme is blended with the linear interpolation scheme in an attempt to maintain both the boundedness and the second-order accuracy of the solution [39]. The LUST scheme is a linear combination of the linear upwind difference scheme (which is stable but first-order) and the central difference scheme with a blending factor γ ($0 \leq \gamma \leq 1$) to optimize the rate of numerical diffusion that will ensure better accuracy and stability.

$$\phi_f = \gamma(\phi_f)_{CDS} + (1-\gamma)(\phi_f)_{LUDS}. \tag{2.63}$$

If $\gamma = 0$, equation (2.63) reduces to a pure linear upwind difference scheme. For standard LUST in the OpenFOAM library, the blending factor has a default value of 0.75, whereas, at the blending factor of 0.9, we get a modified linear interpolation scheme.

Diffusion Term

The diffusion term in equation (2.24) is discretized following the same approach adopted in the previous discretization. Based on the premise of linear variation of ϕ and equation (2.53), it follows that

$$\int_{V_P} \nabla \cdot (\rho\Gamma_\phi \nabla\phi)\, \mathrm{d}V = \sum_f \mathbf{S}_f \cdot (\rho\Gamma_\phi \nabla\phi)_f, \; = \sum_f (\rho\Gamma_\phi)_f \mathbf{S}_f \cdot (\nabla\phi)_f. \tag{2.64}$$

Taking into account the right-hand side of equation (2.64), the actual face value of the scalar $(\rho\Gamma_\phi)$ is estimated by interpolating the cell-centered values of the neighboring cells. When the transported property is the velocity field in an incompressible convection-diffusion flow, the volume integral of the diffusion term reduces to equation (2.64) and the face value of the $(\rho\Gamma_\phi)$ is simplified to $(\rho\nu)$. The other part $(\mathbf{S}_f \mathbf{.} (\nabla\phi)_f)$, however, depends largely on the topology of the grid system and may require special treatment. For computational efficiency, only the elements adjacent to the faces are considered to approximate the gradient of ϕ in the normal direction using the gradient projected in the $\mathbf{d}$ direction. If the grid system is orthogonal, which is rare for most realistic CFD simulations, (i.e., the distance vector $\mathbf{d}$ and face area vector $\mathbf{S}_f$ are parallel) then the dot product can be described, following [68] by

$$\mathbf{S}_f \cdot (\nabla\phi)_f = |\mathbf{S}_f| \frac{\phi_{\mathrm{N}} - \phi_{\mathrm{P}}}{|\mathbf{d}|}. \tag{2.65}$$

If the grid system is non-orthogonal, as commonly seen in applications, the above equation requires additional correction terms to compute the actual diffusion flux on the flow. We can decompose the face area vector, $(\mathbf{S}_f)$, into the orthogonal and non-orthogonal parts

$$\mathbf{S}_f = \mathbf{S}_{f//} + \mathbf{S}_{f\boldsymbol{\Delta}},$$

such that the dot product part of the equation (2.65) is simplified as:

$$\mathbf{S}_f \cdot (\nabla\phi)_f \;\; = \;\; \underbrace{\mathbf{S}_{f//} \cdot (\nabla\phi)_f}_{\text{orthogonal term}} \;\; + \;\; \underbrace{\mathbf{S}_{f\boldsymbol{\Delta}} \cdot (\nabla\phi)_f}_{\text{non–orthogonal correction}}.$$

By decomposing the vector $\mathbf{S}_f$ into the face vectors $\mathbf{S}_{f//}$ and $\mathbf{S}_{f\boldsymbol{\Delta}}$, and requiring $\mathbf{S}_{f//}$ be chosen parallel with the distance vector $\mathbf{d}$, we can apply equation (2.65) on the orthogonal contribution. However, the non-orthogonal part needs to be examined. There are many decompositions in practice. Typical among them are the minimum correction approach, the over-relaxed approach, and the orthogonal correction approach ([39], [68]). The over-relaxed approach has been characterized as the most effective correction approach especially for most transient flow simulations and is implemented in OpenFOAM libraries. Accordingly, using the face area vector decomposition illustrated in Figure 2.7, the over-relaxed correction approach expresses the face vector $\mathbf{S}_{f//}$ as

$$\mathbf{S}_{f//} = \frac{\mathbf{d}}{\mathbf{d} \cdot \mathbf{S}_f} |\mathbf{S}_f|^2,$$

and incorporates a parameter that can regulate the non-orthogonal correction in the event that the control volumes are characterized by a highly non-orthogonal grid.

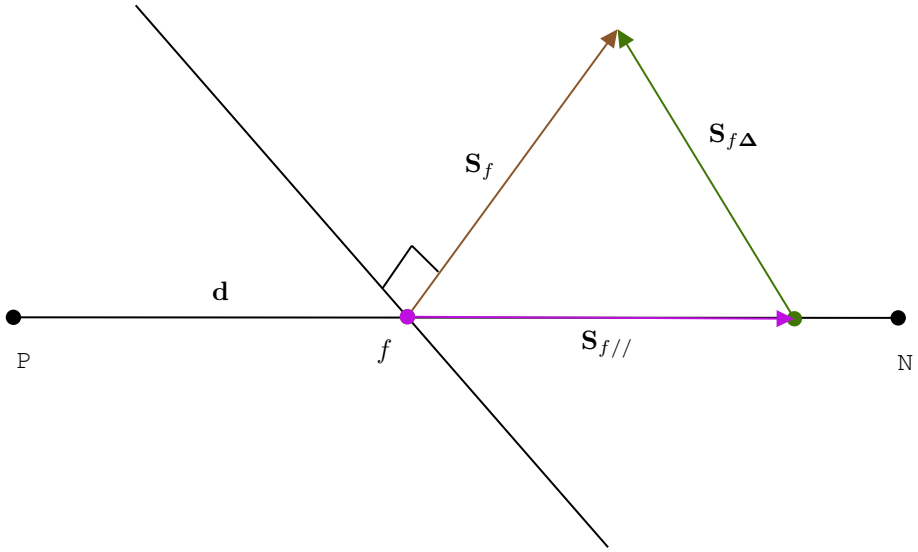


Figure 2.7: Gradient correction for non-orthogonal grid

Source Term

Every other term in equation (2.47) that cannot be identified as convection, diffusion, or temporal terms, can be realized and treated as a source term. The source term Q^ϕ can be a scalar function, which depends on the transport property ϕ. Its discretization takes into consideration possible interaction with other terms of the equation. As a result, its influence on the boundedness and accuracy of the discretization scheme must be considered. Using a simple approach illustrated in Hrvoje [68], the source term can be linearized as

$$Q^\phi = Q_u + \phi Q_p,$$

where Q_u and Q_p are functions of ϕ. Specifically, Q_p is the source term from integrating the pressure gradient over the control volume, and Q_u represents all other source terms. Substituting the above equation into the right-hand side of equation (2.48), the volume integral gives

$$\int_{V_\mathrm{P}} Q^\phi \,\mathrm{d}V = Q_u V_\mathrm{P} + \phi_\mathrm{P} Q_p V_\mathrm{P}.$$

2.5.5 Temporal Discretization

Having discretized the spatial derivatives in equation (2.48) by the finite volume approach, we can next consider the discretization of the temporal term. Temporal discretization entails the process of discretizing the time variable of the problem domain into a finite number of time intervals or discrete steps, by which the derivative of the transient term is evaluated (Figure 2.8). Supposing that the control volume does not change in time and applying spatial discretization to equation (2.48), we obtain a semi-discrete equation of the form

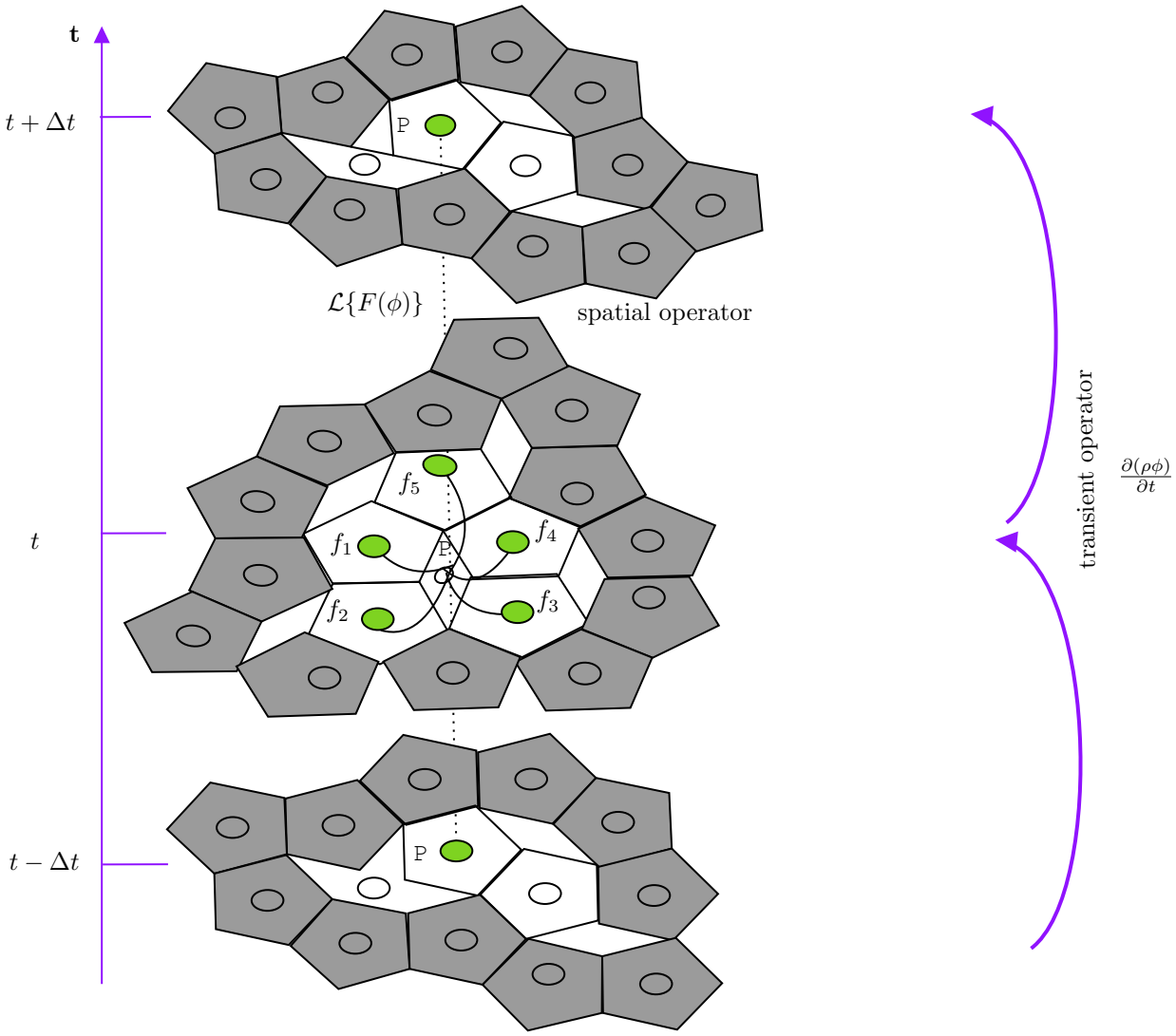


Figure 2.8: Time coordinate, transient, and spatial operator within the control volume [97].

$$\int_t^{t+\Delta t} \left[\frac{\partial}{\partial t} \left(\rho\phi \right)_{\mathrm{P}} V_{\mathrm{P}} \right] \mathrm{d}t = \int_t^{t+\Delta t} \mathbf{F}(\phi)\, \mathrm{d}t,$$

where $\mathbf{F}(\phi)$ serves as a function of transport property ϕ at the cell-centered for all non-transient terms of the Navier–Stokes equations. Then, the transient behavior of any given transport variable [97] such as ϕ in an incompressible transient flow can be described in the form

$$\frac{\partial}{\partial t} \left(\rho\phi \right) + \mathcal{L}\{\mathbf{F}(\phi)\} = 0, \tag{2.66}$$

where $\mathcal{L}\{\mathbf{F}(\phi)\}$ denotes a spatial operator that combines all non-transient terms and $\frac{\partial}{\partial t} \left(\rho\phi \right)$ is the transient term. Using finite volume approach, we first integrate equation (2.66) over the control volume V_{P}, to obtain

$$\int_{V_{\mathrm{P}}} \frac{\partial}{\partial t} \left(\rho\phi \right) \mathrm{d}V + \int_{V_{\mathrm{P}}} \mathcal{L}\{\mathbf{F}(\phi)\}\, \mathrm{d}V = 0,$$

and then apply spatial discretization on the resulting equation about the volume centroid to get

$$\rho V_{\mathrm{P}} \frac{\partial}{\partial t} \left(\phi_{\mathrm{P}} \right) + \mathcal{L}\{\mathbf{F}(\phi_{\mathrm{P}})\} = 0, \tag{2.67}$$

where $\mathcal{L}\{\mathbf{F}(\phi(t))\}$ represents the spatial discretization operator at some reference time t. Algebraically, the spatial discretization operator [97] can be expressed in the form

$$\mathcal{L}\{\mathbf{F}(\phi_{\mathrm{P}})\} = a_{\mathrm{P}}\phi_{\mathrm{P}} + \sum_f (a_f\phi_f - b_{\mathrm{P}}) \equiv F(t, \phi). \tag{2.68}$$

Based on the flow characteristics and domain discretization, many time-stepping schemes have been proposed in the literature to discretize equation (2.66). Following the finite difference method via Taylor series expansion, we take a look at the derivation and the numerical properties of each of the existing time-stepping schemes.

2.5.5.1 Forward Euler Method

Considering equation (2.67), the transient term can be simplified using a Taylor series expansion in a forward fashion about the reference time t. The value of scalar variable ϕ at the time $(t + \Delta t)$ is represented using a Taylor series in terms of the ϕ and its derivatives at the reference time t as

$$\phi(t + \Delta t) = \phi(t) + \Delta t \left(\frac{\partial \phi(t)}{\partial t} \right) + \frac{(\Delta t)^2}{2} \left(\frac{\partial^2 \phi(t)}{\partial t^2} \right) + ...$$

By truncating the expansion after the second term, the first derivative term can be written as

$$\frac{\partial \phi(t)}{\partial t} = \frac{\phi(t + \Delta t) - \phi(t)}{\Delta t} + \mathcal{O}(\Delta t). \tag{2.69}$$

It is obvious that an approximation error is incurred at each time-step as a result of the truncation of the Taylor series, referred to as the local truncation error (LTE) of the method. For the forward Euler method,

the LTE is $\mathcal{O}(\Delta t^2)$, consequently the method is first-order discretization of equation (2.69). Substituting equation (2.69) for the derivative in equation (2.67), yields

$$\frac{\phi_{\mathrm{P}}^{t+\Delta t} - \phi_{\mathrm{P}}^{t}}{\Delta t}\rho V_{\mathrm{P}} + \mathcal{L}\{\mathbf{F}(\phi_{\mathrm{P}}^{t})\} = 0. \tag{2.70}$$

By substituting equation (2.68) into equation (2.70), the complete discretized algebraic equation can be expressed as

$$a_{\mathrm{P}}^{t+\Delta t}\phi_{\mathrm{P}}^{t+\Delta t} + a_{\mathrm{P}}^{t}\phi_{\mathrm{P}}^{t} = b_{\mathrm{P}} - \left(a_{\mathrm{P}}^{t}\phi_{\mathrm{P}}^{t} + \sum_{f} a_f \phi_f^t\right),$$

where

$$a_{\mathrm{P}}^{t+\Delta t} = \rho\frac{V_{\mathrm{P}}}{\Delta t},$$

$$a_{\mathrm{P}}^{t} = -\rho\frac{V_{\mathrm{P}}}{\Delta t},$$

represent the diagonal coefficients obtained as a result of the transient term discretization. The transport property $\phi_{\mathrm{P}}^{t+\Delta t}$ and ϕ_{P}^{t} characterize the transport value at the time level $(t + \Delta t)$ and (t) respectively, and a_{P}, a_f, and b_{P} represent the real coefficients from the spatial discretization.

Characteristics of the forward Euler Method

The forward Euler method is a typical example of an explicit method. An explicit method such as the forward Euler method generates approximate solutions by moving forward in time without solving a system of equations at every time-step [80]. This facilitates easy implementation and generally simplifies the parallelization cost of the computational mesh per unit time-step. However, the time-step size can be limited as a result of the restricted region of stability of an explicit method. As earlier discussed, the notion of stability is beneficial in computational science especially to measure the qualitative and quantitative properties of a given numerical method. Having a stable method is paramount to a bounded solution for a given flow problem. To characterize the stability of the numerical methods, we explore what is known as the absolute stability of time-stepping methods.

Absolute Stability

The classical stability analysis for the numerical time-stepping methods (such as the forward Euler scheme) is examined using the linear model problem [45] of the form

$$\frac{dy}{dt} = \lambda y; \quad y(0) = y_0, \tag{2.71}$$

where λ is a complex constant.

Mathematically, one can easily show that all solutions to the equation (2.71) are stable with respect to small perturbations in the initial condition y_0 if $\Re(\lambda) \leq 0$ and asymptotically stable if $\Re(\lambda) < 0$. Otherwise, the solution is unstable. In this thesis, we examine the absolute stability as a significant way of quantifying the stability of the time-stepping schemes. By applying the forward Euler method to equation (2.71), we

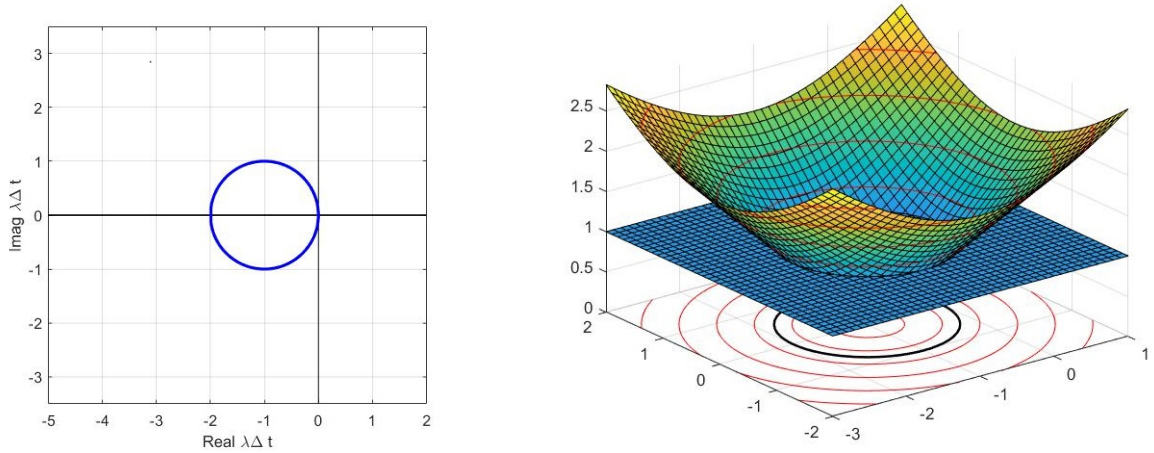
obtain

$$y^{n+1} = R(\lambda\Delta t)y^n,$$

where the amplification factor $R(\lambda\Delta t) = (1 + \lambda\Delta t)$ for the forward Euler method. Hence, we can deduce that the method is absolutely stable [45] if

$$|R(\lambda\Delta t)| \leq 1,$$

otherwise it is unstable. Important parameters are Δt and λ, whose product represents a complex number $z = \lambda\Delta t$ that can be used to characterize the region of absolute stability. Hence, the forward Euler method is stable whenever $|1 + z| \leq 1$. That is, the size of absolute stability of the method is a disk of radius 1 centered at the point $(-1, 0)$ (see Figure 2.9). Figure 2.9 suggests that the forward Euler method is only stable inside the unit circle. This implies that the forward Euler method is conditionally stable, and the size of the computational time-step must be chosen sufficiently small to achieve numerical stability of the desired solution. Few CFD codes have implemented this scheme due to the fact that small time-steps are required to ensure stability. The step size constraint is generally characterized by the Courant–Friedrichs–Levy (CFL) condition [25]. Courant et al. [26] emphasized that "for the numerical solution of a difference equation to converge to the exact solution of the differential equation, the numerical scheme must utilize all the information provided by the initial data."



(a) Absolute region of stability in 2D plane

(b) Absolute region of stability on surface plot

Figure 2.9: The region of absolute stability of forward Euler method on the scalar ODE $\frac{dy}{dt} = \lambda y$ is the interior of the closed curve.

CFL Condition

The CFL condition represents computational condition [29] for the stability of a numerical method. Computationally, the CFL condition for one-dimensional grid can be defined as

$$CFL = \frac{U}{\Delta x}\Delta t,$$

where Δx represents the distance through the grid in the velocity direction, U is taken as the velocity magnitude of the flow, and Δt is the size of the time-step. The CFL condition as a limiting factor represents a dimensionless number, which expresses the ratio of the time-step size to the convective transports. That is, the CFL condition quantifies the rate at which fluid particles (transport quantities) move through each cell in the computational grids. Quantitatively, the CFL condition can be characterized as follows: due to the convective transport, the selected time-step size must be sufficiently small so that the propagation of the transport quantity per unit time-step does not move more than one grid length. If the Courant number is less than one, then the transport property of the flow moves from one cell to another per unit time-step (at most). However, a Courant number greater than one implies that the transport property moves through two or more cells at every time-step and leads to numerical instability. [8].

2.5.5.2 Backward Euler Method

The backward Euler method is otherwise called the implicit Euler method. However, it is simply referred to as Euler in OpenFOAM. Its derivation [97] follows the forward Euler method approach. However, the scalar value of ϕ at the previous time $(t - \Delta t)$ is expanded via a Taylor series expansion in terms of the ϕ and its derivatives at the current time t as

$$\phi(t - \Delta t) = \phi(t) - \Delta t \left(\frac{\partial \phi(t)}{\partial t}\right) + \frac{(\Delta t)^2}{2}\left(\frac{\partial^2 \phi(t)}{\partial t^2}\right) + \dots \tag{2.72}$$

If equation (2.72) is re-arranged as a function of the first derivative, we obtain

$$\frac{\partial \phi(t)}{\partial t} = \frac{\phi(t) - \phi(t - \Delta t)}{\Delta t} + \mathcal{O}(\Delta t). \tag{2.73}$$

When equation (2.73) is substituted into equation (2.67), the discretized equation can be written as

$$\frac{\phi_{\text{P}}^t - \phi_{\text{P}}^{t-\Delta t}}{\Delta t}\rho V_{\text{P}} + \mathcal{L}\{\mathbf{F}(\phi_{\text{P}}^t)\} = 0. \tag{2.74}$$

Applying the algebraic relation in equation (2.68), the complete algebraic form of the discretized equation (2.74) can be given as

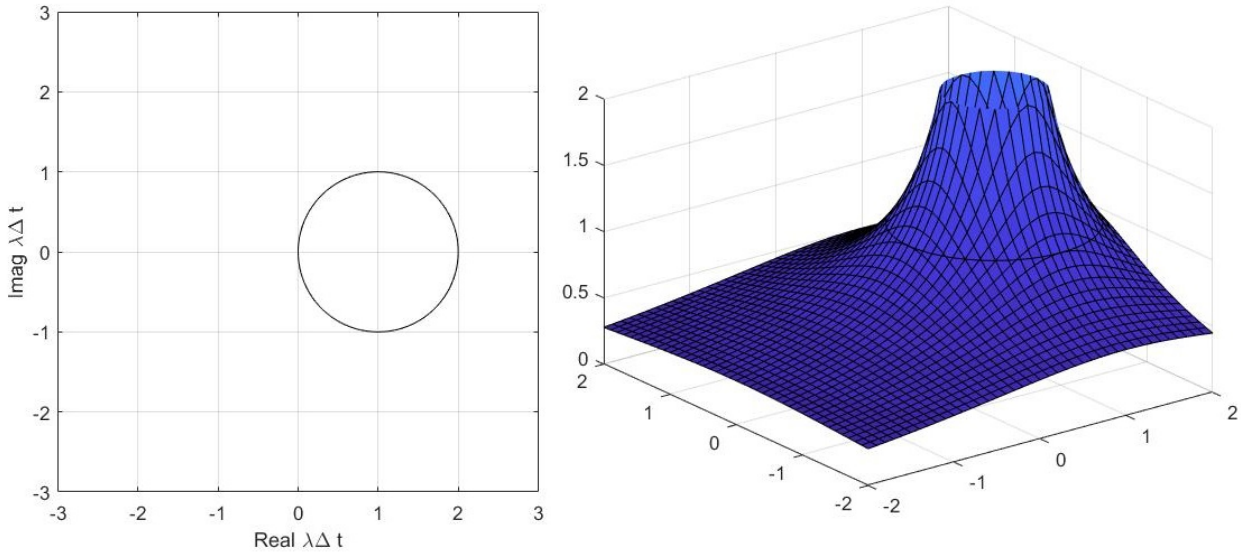
$$\alpha_{\text{P}}^t \phi_{\text{P}}^t + a_{\text{P}} \phi_{\text{P}}^t + \sum_f a_f \phi_f^t = b_{\text{P}} + a_{\text{P}}^{t-\Delta t} \phi_{\text{P}}^{t-\Delta t},$$

where

$$\alpha_{\text{P}}^t = \rho \frac{V_{\text{P}}}{\Delta t},$$
$$a_{\text{P}}^{t-\Delta t} = -\rho \frac{V_{\text{P}}}{\Delta t}.$$

Characteristics of the backward Euler Method

The evaluation of the spatial operator and the new temporal coefficients at the same time level produces a system of algebraic equations that may take a longer time to solve. The solution of a system of algebraic equations at each time-step is a peculiar attribute of all classes of an implicit method. The backward Euler method has $\mathcal{O}(\Delta t)$ global accuracy, just like the forward Euler method. Unfortunately, implicit methods are generally more difficult to implement than explicit methods. However, the backward Euler method is unconditionally stable (see Figure 2.10).



(a) Absolute region of stability in 2D plane

(b) Absolute region of stability on surface plot

Figure 2.10: The region of absolute stability of backward Euler method on the scalar ODE $\frac{dy}{dt} = \lambda y$ is the exterior of the closed curve.

2.5.5.3 Backward Differentiation Formula

The backward differentiation formula of order k (BDFk) is one of the cost-effective methods for the numerical integration of stiff differential equations [1]. The BDFk method is a family of an implicit linear multi-step methods based on a Taylor series expansion. Quantitatively, the BDFk method offers an approximation to the derivative of the unknown variable ϕ at the reference time t in terms of its function values $F(\phi)$ at the current and previous time-steps. In respect of equation (2.66)

$$\frac{\partial}{\partial t}\left(\rho\phi\right) + \mathcal{L}\{\mathbf{F}(\phi)\} = 0, \tag{2.75}$$

the general formula for the BDFk method [40] can be written as

$$\sum_{j=0}^{k} a_j \left(\rho\phi\right)_{n+j} = \Delta t \beta_k \mathbf{F}_{n+k},$$

where Δt represents the time-step size, $a_k = 1$, a_j, $j = 1, \cdots, k$, β_k are unknown constant coefficients of the method. With the evaluation of F at every time-step for the unknown scalar variable ϕ, the BDFk method generates a system of nonlinear equations that must be solved at each time-step. The method coefficients a_j and β_k are distinctly determined so that the method achieves order k, which is the maximum possible. The BDFk method is distinguished according to the order of the formula. If $k = 1$, we have the backward differentiation formula of first-order, or simply BDF1, which corresponds to the backward Euler method. When $k = 2$, we have the BDF2 method. The regions of absolute stability for the BDF methods are illustrated in Figure 2.11. The figure indicated that the region of the absolute stability diminishes as k increases but remains unbounded for $k \leq 6$.

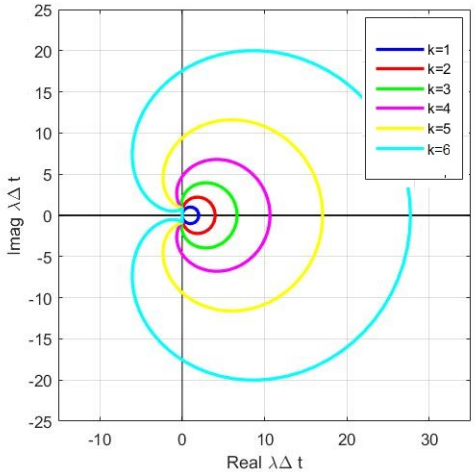


Figure 2.11: The region of absolute stability of the BDFk methods ($k = 1, 2, \ldots, 6$) on the scalar ODE $\frac{dy}{dt} = \lambda y$ is the exterior of the corresponding closed curve.

By numerical analysis, the BDF2 method is second-order accurate and has a better stability property for the numerical integration of stiff differential equations than the forward Euler method. The BDF2 method can be derived by expanding the value of ϕ at $(t - \Delta t)$ and $(t - 2\Delta t)$ respectively using a Taylor series expansion about t, to obtain

$$\phi(t - \Delta t) = \phi(t) - \Delta t \left(\frac{\partial \phi(t)}{\partial t} \right) + \frac{(\Delta t)^2}{2} \left(\frac{\partial^2 \phi(t)}{\partial t^2} \right) + \mathcal{O}(\Delta t^3), \tag{2.76}$$

$$\phi(t - 2\Delta t) = \phi(t) - 2\Delta t \left(\frac{\partial \phi(t)}{\partial t} \right) + \frac{4(\Delta t)^2}{2} \left(\frac{\partial^2 \phi(t)}{\partial t^2} \right) + \mathcal{O}(\Delta t^3), \tag{2.77}$$

If we eliminate the second derivative term and re-arrange the resulting expressions, we obtain an expression for the first derivative as

$$\frac{\partial \phi(t)}{\partial t} = \frac{3\phi(t) - 4\phi(t - \Delta t) + 2\phi(t - 2\Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2). \tag{2.78}$$

By this derivation, we can deduce that the BDF2 method has $\mathcal{O}(\Delta t^2)$ accuracy. Substituting equation (2.78)

into equation (2.67), we obtain

$$\left(\alpha_{1,\mathrm{P}}^{t} + a_{\mathrm{P}}^{t}\right)\phi_{\mathrm{P}}^{t} + \sum_{f} a_f \phi_f^{t} = b_{\mathrm{P}} - a_{P}^{t-\Delta t}\phi_{\mathrm{P}}^{t-\Delta t} - a_{\mathrm{P}}^{t-2\Delta t}\phi_{\mathrm{P}}^{t-2\Delta t},$$

where

$$\alpha_{1,\mathrm{P}}^{t} = \rho\frac{3V_{\mathrm{P}}}{2\Delta t},$$
$$a_{\mathrm{P}}^{t-\Delta t} = -\rho\frac{2V_{\mathrm{P}}}{\Delta t},$$
$$a_{\mathrm{P}}^{t-2\Delta t} = \rho\frac{V_{\mathrm{P}}}{2\Delta t}.$$

The BDF2 method is implemented in OpenFOAM 2.2.0 and versions thereafter, with usage name –backward in the ddt scheme of the `fvScheme` (see Appendix A).

2.5.5.4 Crank–Nicolson Method

Due to the inherent limitations with explicit time-stepping methods, Crank and Nicolson [27] designed a more reliable time-stepping scheme called the Crank–Nicolson (or trapezoidal rule) method. The Crank–Nicolson method presented a more accurate approximation of a derivative term by incorporating a weighted average of the second spatial step at the time $(t + \Delta t)$ and $(t - \Delta t)$. Expanding the transport property ϕ at the time $(t + \Delta t)$ and $(t - \Delta t)$ via a Taylor series expansion, we obtain

$$\phi(t + \Delta t) = \phi(t) + \Delta t\left(\frac{\partial\phi(t)}{\partial t}\right) + \frac{\Delta t^2}{2}\left(\frac{\partial^2\phi(t)}{\partial t^2}\right) + \frac{\Delta t^3}{6}\left(\frac{\partial^3\phi(t)}{\partial t^3}\right) + \ldots, \tag{2.79}$$

$$\phi(t - \Delta t) = \phi(t) - \Delta t\left(\frac{\partial\phi(t)}{\partial t}\right) + \frac{\Delta t^2}{2}\left(\frac{\partial^2\phi(t)}{\partial t^2}\right) - \frac{\Delta t^3}{6}\left(\frac{\partial^3\phi(t)}{\partial t^3}\right) + \ldots. \tag{2.80}$$

To obtain an expression for the first derivative term, we subtract equation (2.80) from (2.79) and get

$$\frac{\partial\phi(t)}{\partial t} = \frac{\phi(t + \Delta t) - \phi(t - \Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2). \tag{2.81}$$

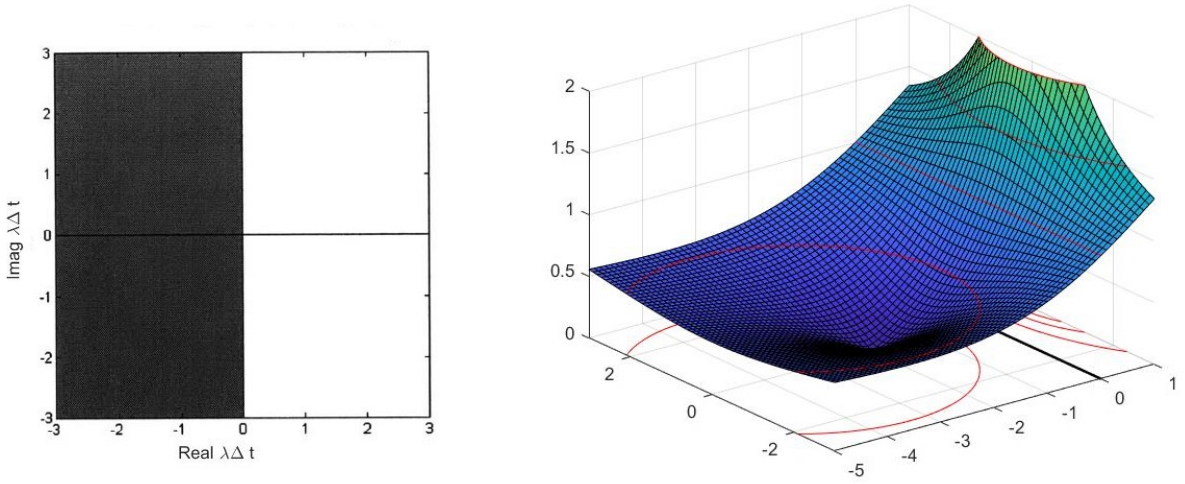
Quantitatively, the order of accuracy of the Crank–Nicolson method is $\mathcal{O}(\Delta t^2)$. That is, the Crank–Nicolson scheme is more accurate and less restrictive than the forward Euler method. In addition, the Crank–Nicolson method is absolutely stable (see Figure 2.12) but conditionally bounded [58]. The unconditional stability of the Crank–Nicolson method is, however at the expense of solving the resulting system of the algebraic equations at every new time-step, and the solution can still suffer from some spurious oscillations.

The substitution of equation (2.81) into equation (2.67) yields

$$\frac{\phi^{t+\Delta t} - \phi^{t-\Delta t}}{2\Delta t}\rho V_{\mathrm{P}} + \mathcal{L}\{\mathbf{F}(\phi_{\mathrm{P}}^{t})\} = 0. \tag{2.82}$$

Introducing the algebraic equation (2.68) into equation (2.82), the complete discretized form of the transport equation is obtained as

$$\alpha_{\mathrm{P}}\phi_{\mathrm{P}}^{t} = b_{\mathrm{P}} - \left(a_{\mathrm{P}}\phi_{\mathrm{P}}^{t-\Delta t} + \sum_{f} a_f \phi_f^{t-\Delta t}\right) - a_{\mathrm{P}}^{t-2\Delta t}\phi_{\mathrm{P}}^{t-2\Delta t}, \tag{2.83}$$



(a) Absolute region of stability in 2D plane

(b) Absolute region of stability on surface plot

Figure 2.12: Region of absolute stability of the Crank–Nicolson method on the scalar ODE $\frac{dy}{dt} = \lambda y$.

where

$$\alpha_{\mathrm{P}} = \frac{\rho V_{\mathrm{P}}}{2\Delta t},$$
$$a_{\mathrm{P}}^{t-2\Delta t} = -\frac{\rho V_{\mathrm{P}}}{2\Delta t}.$$

Suppose we modify equation (2.83) by the linear approximation of $\phi(t - \Delta t)$ using

$$\phi(t - \Delta t) \approx \frac{\phi(t) + \phi(t - 2\Delta t)}{2},$$

then equation (2.83) can be written as

$$\alpha_{\mathrm{P}}\phi_{\mathrm{P}}^{t} + \frac{1}{2}\left(a_{\mathrm{P}}\phi_{\mathrm{P}}^{t} + \sum_{f} a_f \phi_f^{t}\right) = b_{\mathrm{P}} - \frac{1}{2}\left((a_{\mathrm{P}} + 2a_{\mathrm{P}}^{t-2\Delta})\phi_{\mathrm{P}}^{t-2\Delta t} + \sum_{f} a_f \phi_f^{t-2\Delta t}\right), \tag{2.84}$$

which requires the face and cell values of ϕ at the current and two previous time-steps.

2.5.5.5 TR–BDF2 Method

As far as the time-stepping scheme accuracy is concerned, the Crank–Nicolson method is still a popular method for solving differential equations [27]. The Crank–Nicolson method is generally efficient, except for extremely stiff problems (such as encountered in a fluid flow) [54]. Therefore, having a time-stepping scheme with better stability properties as well as second-order accuracy is highly desirable.

With this in mind, the TR–BDF2 is a possible choice. The TR–BDF2 method is a composite scheme proposed by Bank et al. [10] and combines one stage of the trapezoidal rule followed by the second-order BDF2 method. The combination is empirically justified on the premise of integrating the good accuracy of the trapezoidal rule with the stability and damping of fast modes assured by the BDF2 method. The TR–BDF2 method is

essentially designed to attenuate spurious oscillation related to the application of the Crank-Nicolson method on a stiff system of ODEs. The TR–BDF2 stages are optimized so that both the trapezoidal and BDF2 stages use the same Jacobian matrix in the evaluation of the unknown function at each time-step.

The TR–BDF2, as a family of fully implicit Runge Kutta methods, has a better stability property than A–stability property of the Crank–Nicolson method [86]. This appealing feature qualifies the TR–BDF2 method to handle the transient term in equation (2.67) or any temporal discretization of PDEs.

To derive the TR–BDF2 method, we integrate the semi–discretized equation (2.67) from t to $(t+\Delta t)$. After the integration, we apply the trapezoidal rule on the resulting integral terms to accelerate the solution from t to $(t+\gamma\Delta t)$ and obtain

$$\rho\Big[\phi^{t+\gamma\Delta t}-\phi^{t}\Big]=\gamma\frac{\Delta t}{2}\Big[F(t,\phi^{t+\gamma\Delta t})+F(t,\phi^{t})\Big]. \tag{2.85}$$

Thereafter, the BDF2 is applied to equation (2.85) to advance the solution from $(t+\gamma\Delta t)$ to $(t+\Delta t)$ and get

$$\rho\phi^{t+\Delta t}-\frac{1}{\gamma(2-\gamma)}\rho\phi^{t+\gamma\Delta t}+\frac{(1-\gamma)^2}{\gamma(2-\gamma)}\rho\phi^{t}=\frac{(1-\gamma)}{(2-\gamma)}\Delta tF(t,\phi^{t+\Delta t}), \tag{2.86}$$

where $0<\gamma<1$. As a family of diagonally implicit Runge–Kutta (DIRK) methods, the Butcher tableau for the TR–BDF2 method is presented in Table 2.3, with $\gamma=2-\sqrt{2}$, $\alpha=\frac{\gamma}{2}$, and $a=\frac{\sqrt{2}}{4}$. Even though there are 2 stages in the TR–BDF2 method, it is still a one-step method with a simple three-point average.

0	0		
γ	α	α	
1	a	a	$\frac{1-\gamma}{2-\gamma}$
	a	a	$\frac{1-\gamma}{2-\gamma}$

Table 2.3: Butcher tableau for the TR–BDF2 method [17].

From Table 2.3, it can be seen that the TR–BDF2 method has First Same As Last (FSAL) property [60]. That is, the first stage of any step is identical to the last stage of the preceding step such that in any step, the first explicit stage need not be computed. This attribute is a major characteristic feature of the TR–BDF2 method from a standard BDF2 method, which required two previous time–steps evaluation. According to Bank et al. [10], the local truncation error of the TR–BDF2 method can be minimized when γ is set to $2-\sqrt{2}$, so that the two stages can have the same Jacobian matrix.

Other Stability Related Concepts

A–Stable Method

A time-stepping method for the problem (2.71) is A–stable [80] if the condition

$$\|y^{n+1}\|\leq\|y^{n}\|$$

holds, and the absolute stability region spans the entire domain of the left half plane. Thus, explicit time-stepping methods are not A–stable because they have finite absolute stability region. For example, the

A–stable property of the TR–BDF2 method is examined by applying first the Crank–Nicolson (TR) method to the test problem (2.71) to obtain

$$y^{n+1} = R(\lambda\Delta t)y^n,$$

where $R(\lambda\Delta t) = \frac{1+\frac{1}{2}(\lambda\Delta t)}{1-\frac{1}{2}(\lambda\Delta t)}$. Then for the absolute stability, we have

$$|R(\lambda\Delta t)| \leq 1,$$

and the region of absolute stability encloses the whole domain of the left half plane (see Figure 2.12). Therefore, the CN method is A–stable. In a similar approach, we can show that the BDF2 method is A–stable. Although numerical methods with A–stable property are classic methods to get a stable solution, sometimes they fail to provide adequate damping to mitigate instability, particularly for complex flow problems. Round off errors can readily accumulate to the point where they induce an unbounded solution.

L–Stable Method

Qualitatively, A–stable property is not sufficient to ensure a stable solution for all categories of problems. A more desirable characteristic for a given numerical method is L–stability, which has a stronger requirement than A–stability. A time-stepping method is L–stable [80] if it is A–stable and

$$\lim_{|z|\longrightarrow\infty} \frac{\|y^{n+1}\|}{\|y^n\|} = 0.$$

By this definition, we show that the CN method is A-stable but not L-stable. Using a model problem defined by equation (2.71), we have that

$$\frac{y^{n+1}}{y^n} = \left|\frac{1+\frac{1}{2}(\lambda\Delta t)}{1-\frac{1}{2}(\lambda\Delta t)}\right| = \frac{1+\Delta t\mathrm{Re}(\lambda)+\frac{1}{4}\Delta t^2|\lambda^2|}{1-\Delta t\mathrm{Re}(\lambda)-\frac{1}{4}\Delta t^2|\lambda^2|} \longrightarrow 1 \text{ as } \lambda\Delta t \longrightarrow \infty.$$

In a similar way, we can show that the backward Euler and TR–BDF2 methods are L–stable. In the step that follows, we establish that the TR–BDF2 method is L–stable. For the trapezoidal stage of the TR–BDF2 method, we get

$$y^{n+\gamma} = \frac{2-(\gamma\lambda\Delta t)}{2+(\gamma\lambda\Delta t)}y^n,$$

which is substituted into the BDF2 stage, to obtain

$$(2-\gamma)y^{n+1} - (1-\gamma)\lambda\Delta t y^{n+1} = \frac{1}{\gamma}\frac{2-(\gamma\lambda\Delta t)}{2+(\gamma\lambda\Delta t)}y^n - \frac{(1-\gamma)^2}{\gamma}y^n.$$

By setting $z = \lambda\Delta t$, the above expression is simplified as

$$y^{n+1} = R(z)y^n, \tag{2.87}$$

where the amplification factor $R(z) = \frac{2(2-\gamma)-z(1+(1-\gamma)^2)}{z^2(\gamma-1)\gamma+z(-\gamma^2+4\gamma-2)+2(2-\gamma)}$. The amplification factor value tends to zero as $|z| \longrightarrow \infty$ implying that the TR–BDF2 method is L–stable [17]. The TR–BDF2 region of absolute stability is illustrated in Figure 2.13c.

For an easy implementation of the TR–BDF2 method in OpenFOAM, we have approximated the first term on the right-hand side of equation (2.85) with

$$F(t, \phi^{t+\gamma\Delta t}) = (1-\gamma)F(t, \phi^{t}) + \gamma F(t, \phi^{t+\Delta t}),$$

so that at the second stage of the scheme, the modified TR–BDF2 method becomes

$$\rho\phi^{t+\Delta t} + \frac{(1-\gamma)^2 - 1}{\gamma(2-\gamma)}\rho\phi^{t} = \frac{\Delta t}{2}\Big[F(t, \phi^{t+\Delta t}) + F(t, \phi^{t})\Big].$$

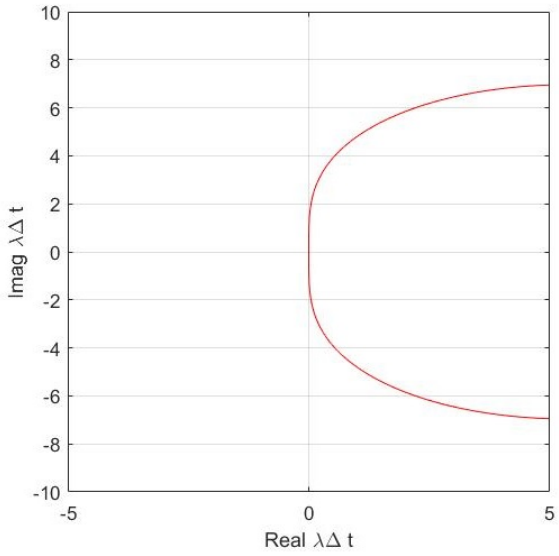
The OpenFOAM implementation of the modified TR–BDF2 method is given in Appendix A. However, for the modified TR–BDF2 method, the amplification factor is

$$R(z) = \frac{z\gamma(2-\gamma) + 2(1-(1-\gamma)^2)}{\gamma(2-z)(2-\gamma)}.$$

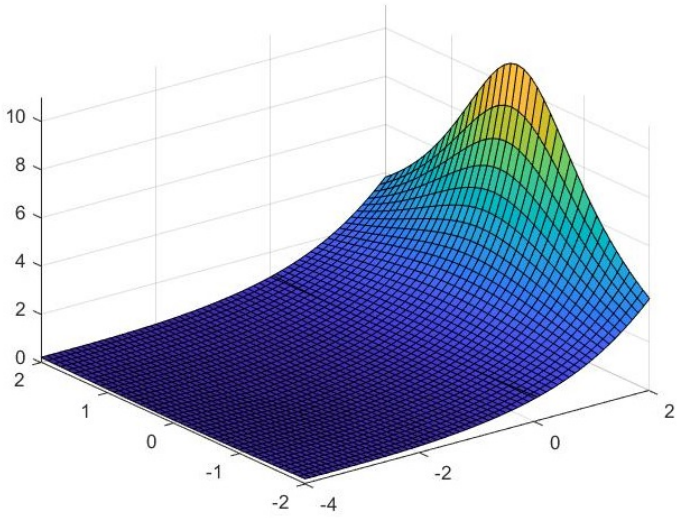
with

$$\lim_{|z|\longrightarrow\infty} R(z) = -1.$$

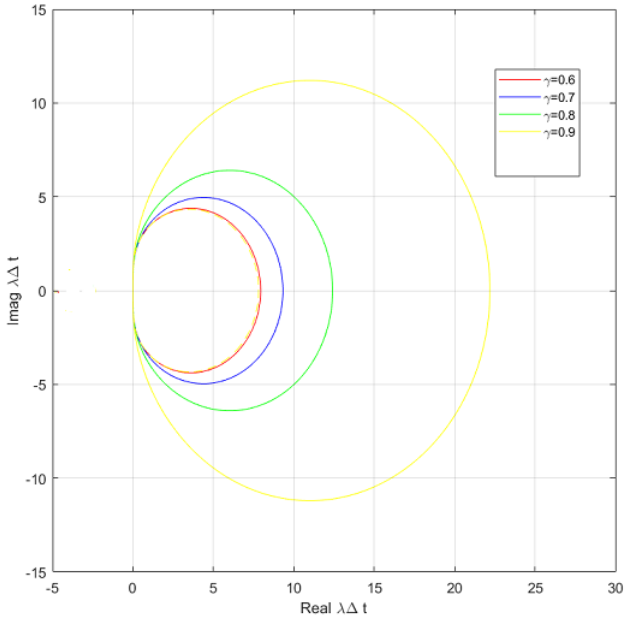
suggesting that the modified TR–BDF2 method is not L-stable.



(a) Absolute region of stability in 2D plane



(b) Absolute region of stability on surface plot



(c) The TR–BDF2 region of absolute stability at different values of γ is the exterior region of the corresponding closed curve.

Figure 2.13: The region of absolute stability of the TR–BDF2 method on scalar ODE $\frac{dy}{dt} = \lambda y$ is the exterior of the closed curve.

2.6 Closure

The central goal of turbine aerodynamic simulation is to evaluate the induced velocity field, pressure distribution field, and the performance characteristics of the turbine system. One good way to achieve this goal is through CFD. The geometrical features of the turbine system and the corresponding flow governing equations can be studied and easily analyzed using the CFD techniques.

The unsteady flow around the Lux VAWT is characteristically simplified by an incompressible, continuum, and isothermal Navier–Stokes equations. These equations along with the turbulent model are derived from the principle of mass and momentum conservation in the flow system. The URANS technique is applied to predict the turbulent in the flow, and the finite volume discretization is adopted to discretize the governing equation (2.12) subject to the specific boundary conditions (2.13)–(2.18) in the domain of interest. The flow transport property such as pressure and velocity fields is computationally predicted by solving the corresponding system of PDEs using different discretization schemes.

The simulation geometry illustrated in Figure 2.1 is characterized by the following preliminary dimensions. The Lux turbine blades (which are positioned inside the rotating domain) is 18.3 m and 29.3 m in diameter and height respectively. The rotating domain (which is cylindrical in shape) is 27.5 m in diameter and 58.6 m high. Whereas the outer domain (i.e., the rectangular cuboid), which represents the wind tunnel, is 82.4 m wide by 183 m long and 73.2 m high.

The simulation follows the Reynolds-Averaged Navier–Stokes technique and adopts the k–ω SST turbulent model (2.35)–(2.37) to estimate the turbulent effect on the flow. The resulting system of differential equations is resolved using different time-stepping methods and pressure-based solvers. The qualitative properties of the numerical schemes were further examined for the best characteristic performance. The next chapter describes the meshing procedures for the Lux VAWT geometry, the discretization schemes implementation, and pressure velocity coupling algorithms as used in OpenFOAM.

3 Methodology

In the preceding chapter, a transport equation discretization process was described. With this discretization, a number of numerical schemes were presented to solve the governing equations. Herein, the CFD toolkit called OpenFOAM and its application for the numerical simulation of the Lux VAWT are described in order to realize our research objectives, which are to model a turbulent flow around the Lux VAWT, evaluate its average power output, and compare the performance characteristics with experimental data.

3.1 OpenFOAM

Open source Field Operation And Manipulation (OpenFOAM) is a cost-free, open–source CFD package developed primarily by the OpenCFD Ltd [100] to resolve flows numerically and provide different solvers, libraries, and utilities for various CFD problems. The OpenFOAM package offers a robust and adaptable simulation platform and incorporates a C++ toolbox for the development of personalized numerical solvers and pre-/post-processing utilities shown in Figure 3.1 for the solution of continuum mechanics problems. C++ , as an object oriented programming language, was a good choice because of its highest modularity.

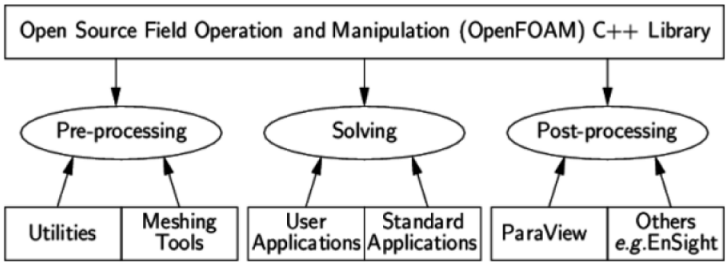


Figure 3.1: OpenFOAM compartmental structure [100]

Today, OpenFOAM has a broad range of features and versions to simulate most industrial turbulent flows in automotive aerodynamics, combustion processes, chemical reactions, heat transfer, liquid sprays, and films ([68], [97]). It is possible to modify the default codes or develop new program modules (solvers) entirely using basic programming skills. The present work uses OpenFOAM 3.0.x, which was released in 2015, due to its flexibility in incorporating any turbulence simulation types and on the basis that:

- It is popular in industry with many examples on practical projects that can be accessed online.

- It uses command–line tools and works well with scripts.
- It has a flexible and friendly syntax for partial differential equations and feature great deal of unstructured polyhedral mesh capabilities.
- Its applications on some industrial CFD problems generate results that were found comparable to general-purpose commercial closed-source CFD packages.

Nevertheless, OpenFOAM is less intuitive than all available commercial professional CFD packages, primarily because of the absence of an integrated graphical user interface. Like many open-source CFD toolkits, OpenFOAM is not properly documented and requires a great deal of technical know-how. It also has a steep learning curve owing to the substantial number of tools and configuration files required to run an end-to-end simulation.

A CFD simulation with OpenFOAM requires a case folder that contain all the relevant files to perform the simulation including system, constant, and time directories as shown in Figure 3.2. In this thesis, the simulation directories contain useful files and numerical schemes to successfully run a given VAWT simulation with OpenFOAM. For example, a constant directory holds the material (physical) properties of the fluid, turbulence properties, and mesh information. The solution controls setting, discretization schemes, and time step controls are defined in the system directory, whereas the time directory (sometimes called the 0 directory) holds individual files of data for a particular variable field. Each time file is labeled, in conformity with the simulated time at which simulation data were reported. If the simulation starts at time $t = 0$, then we create a 0 directory to contain all the initial flow fields to start the simulation.

3.2 Pre–processing

Pre–processing is the first requisite step in generating the CFD solution [74]. This process consists of building the geometry of the VAWT under investigation as well as generating a mesh system that will consequently be converted to a readable format for the solver. Pre–processing is often the most demanding and tedious step that must be done with precision to achieve a convergent and stable CFD solution. The accuracy of any CFD solution is dependent on the features of the mesh used to carry out the simulations [100]. In CFD, there are two types of mesh, namely structured and unstructured [6].

A structured mesh typically consists of regular repeating elements, generally characterized by quadrilateral elements in 2D or hexahedral elements in 3D [75]. These elements are assembled in a uniform repeating pattern so that the regular connectivity information of the elements is stored implicitly to minimize the computational effort during the simulation. However, generating a structured mesh is challenging task especially for complex geometries, such as the Lux VAWT.

On the other hand, an unstructured mesh is a collection of an arbitrary shaped elements, mostly tetrahedra, with an explicitly defined connectivity ([74], [75]). All the mesh nodes are collectively defined to

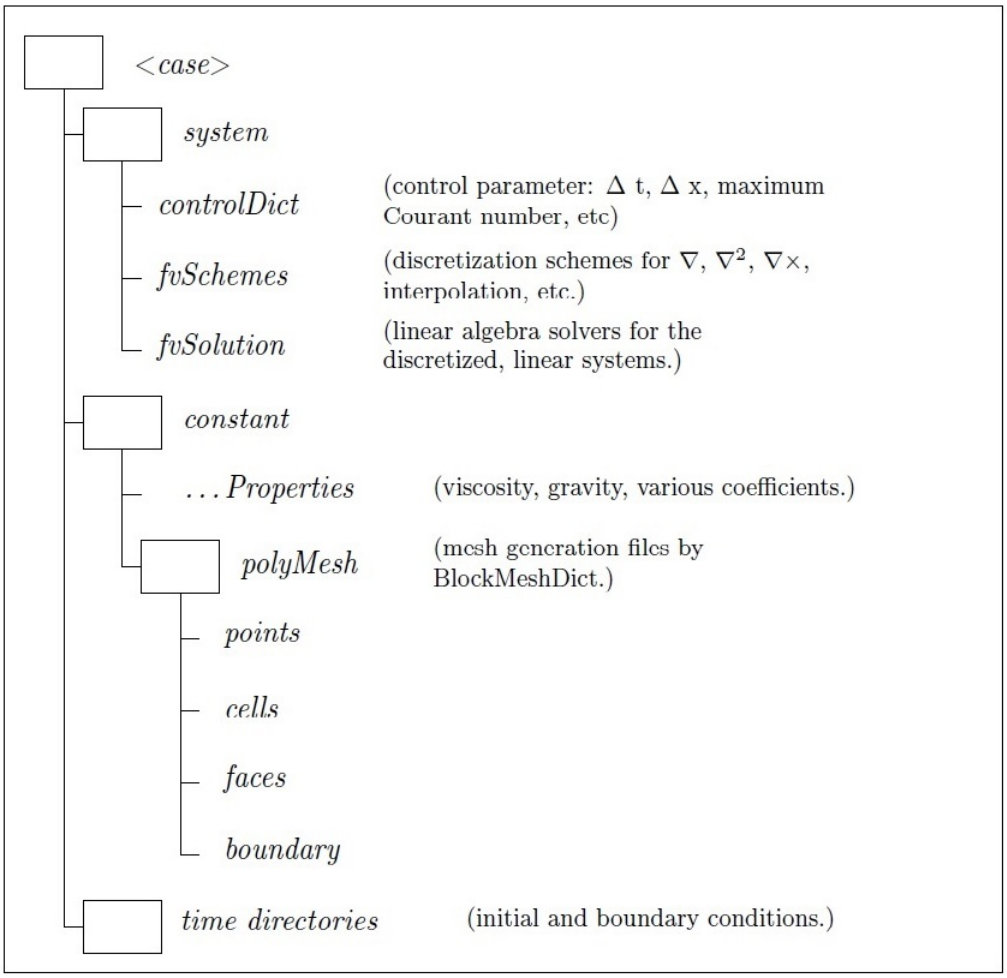


Figure 3.2: OpenFOAM case structure [100]

automate the meshing process and with no prior or specified pattern. The unstructured mesh generation entails two basic phases, point creation and definition of connectivity between these points. As a result, large computational memory is required to store the elements, nodes, and the connectivity pattern between the points. Unstructured meshing is usually characterized with flexible and adaptive mesh refinement; however, this can lead to skewed elements that may adversely affect the computational accuracy. An unstructured mesh is mostly applied in the finite element and finite volume methods because of their ability to resolve complex physical geometries. For the Lux VAWT simulation, the computational meshes were unstructured with the tetrahedral mesh elements.

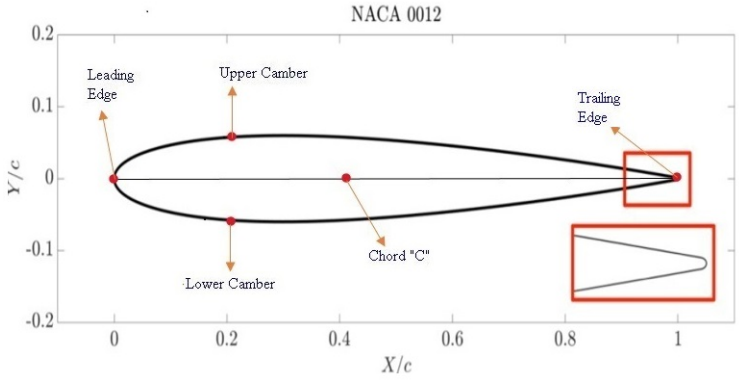
3.2.1 Lux Model Geometry

The key geometrical attributes and operational characteristics of the Lux VAWT are presented in Table 3.1. The Lux VAWT generates a power output of 50.0 kW at a favorable wind speed. The experimental data presented in Appendix C indicate that the Lux VAWT starts to generate practical power at a wind speed (cut-in speed) of 4 m/s, with the rated capacity (cut-out speed) attained at about 25 m/s. The Lux VAWT has 6 curve blades primarily designed from the symmetrical airfoil NACA 0012 with a chord size of 0.2 m and measures 29.3 m high by 18.3 m in diameter. The Lux VAWT is novel in its rotor design and favors lower wind speeds associated with urban settings.

The simplified Lux VAWT geometry (see Figure 3.3) were designed with help of the ANSYS DesignModeler and SolidWorks. The 3D geometry consists of airfoil NACA 0012 extruded in 3D to create a curve egg–like shape displayed in Figure 3.3b, the rotating domain (i.e., the cylindrical inner zone) that holds the six blades and rotor, and the outer domain that represents virtually the air–flow around the Lux VAWT. The outer domain is stationary and represents the wind tunnel. The Lux VAWT parts were assembled together in Figure 3.4 to form the computational domain for the simulation. The physical dimensions of the simulating geometries are defined in Table 3.2 with H, W, L, L_i, and L_c representing the height, the width, the length, the inlet distance to the center of the turbine, and the rotor distance to the exit plane of the computational domain, respectively. These large dimensions are necessary to minimize the wind tunnel blockage effect and prevent the occurrence of reverse flow at the outlet patch [108]. The cylindrical rotating domain has an initial diameter of $1.5D$. The wind tunnel and rotor inner zone are separated by the turbine interface region to ensure continuity in meshing and flow field computation.

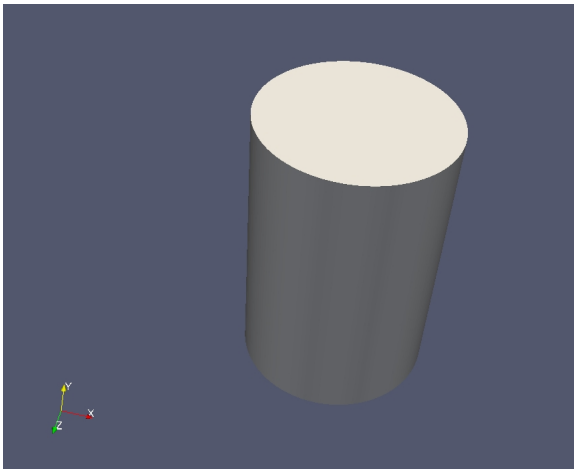
Number of rotors	1
Number of blades	6
Type of airfoil	NACA0012
Airfoil chord	0.2 m
Reynolds number	$133,000$ – $800,000$
Cut in wind speed	4 m/s
Cut out wind speed	25 m/s
Swept area	360 m^2
Rotor diameter D	18.3 m
Rotor height H_r	29.3 m
Angle of attack	$8°$
Rotational speed	30 – 40
Tip speed ratio λ	1.74 – 7.19

Table 3.1: The design features of the Lux VAWT.

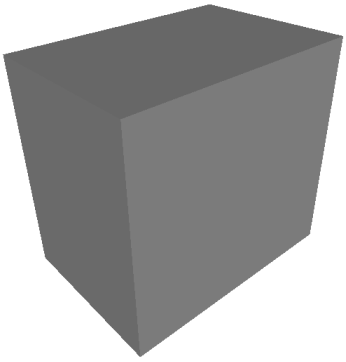


(a) Description of airfoil section design parameters in 2D [55].

(b) STL file of the Lux blades in 3D



(c) Rotating domain of the Lux VAWT geometry in 3D



(d) Stationary domain in 3D

Figure 3.3: Model Geometry

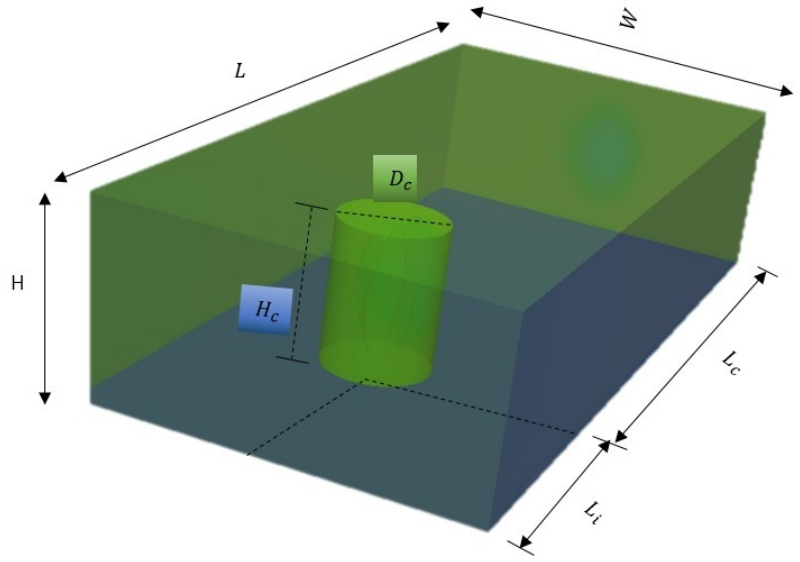


Figure 3.4: Assembled Geometry and dimensions for the Lux VAWT

Dimension	Geometry A		Geometry B	
Label	Domain Dimension	Abs. Dimensions	Domain Dimension	Abs. Dimensions
Cylinder Diameter D_c	1.5D	27.45	2D	36.6
Cylinder Height H_c	$2H_r$	58.6	$2.5H_r$	73.25
H	4 D	73.2	6D	109.8
W	4.5D	82.35	5D	91.5
L	10D	183	11D	201.3
L_i	2.5D	45.75	3D	54.9
L_c	7.5D	137.25	8D	146.4

Table 3.2: Geometrical dimensions in meters for the Lux VAWT.

3.2.2 Mesh Generation Process

A pre-processing step for the CFD simulation involves the discretization of the VAWT geometrical surfaces and is called mesh generation [75]. Mesh generation plays a critical role on the efficiency and the accuracy of the computation [74]; element or cell shape and size often characterize both the quality of the computational results and the amount of computational work (e.g., number of iterations) required to attain a convergent solution.

Generating a mesh with high quality may depend on the complexity of the geometry and the user experience with meshing utilities. Many of the CFD meshing utilities are embedded and automated, and there is a

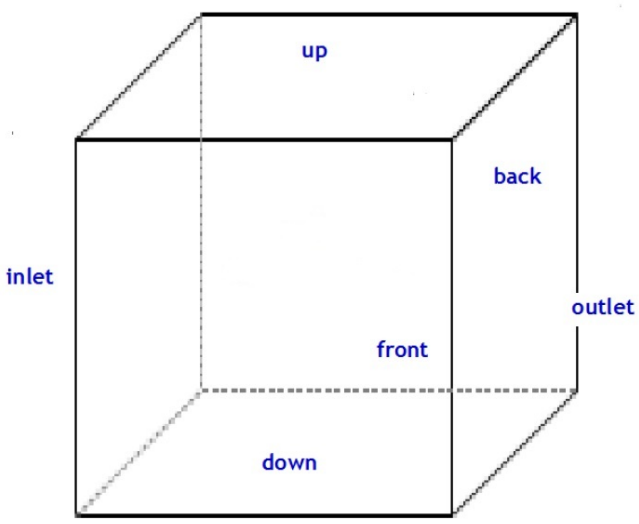
growing attraction towards automated mesh generation [8], where a meshing algorithm regulated by pre-defined parameters is applied to a pre-existing CAD geometry for optimal mesh quality. Most modern mesh generating packages, such as OpenFOAM `SnappyHexMesh` and ANSYS Mesher, have a standard algorithm to generate the structured and arbitrary unstructured or polyhedral meshes that allow for a broad range of cell shapes for most complex geometries.

Mesh quality is defined by the CFD simulation requirements. The acceptable level of mesh quality depends on the characteristic requirement of the applied turbulence modeling. For instance, the turbulence modeling using LES requires that mesh quality satisfies a minimum cell size to resolve turbulent eddies, thus requiring higher level of mesh quality than the RANS technique. Although there is no single standard metric to assess the quality of a mesh, there are some basic practices to follow, and the minimum quality criteria such as skewness, mesh non-orthogonality, aspect ratio, and smoothness must be in mind when generating a mesh system for the VAWT simulation.

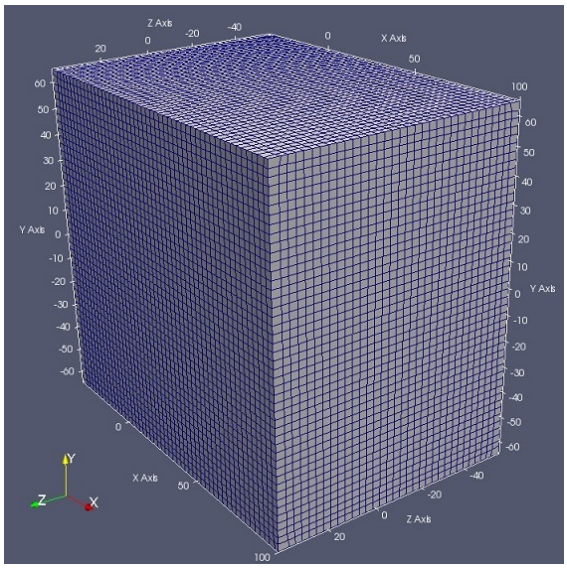
There are two major utilities for mesh generation in the OpenFOAM software package. One is through the `blockMesh` utility, which generates elementary meshes of blocks with hexahedral cells. The second utility is the `SnappyHexMesh` tool, which is used primarily to generate intricate 3-dimensional meshes of hexahedral and split–hexahedral cells from the triangulated surface geometries in stereolithography (STL) format. The STL geometry surface represents a triangulated surface approximation of the physical geometry and includes only information about the triangle corner coordinates. The domain of the Lux VAWT geometry given in Figure 3.4 incorporates one main block (outer domain) and the six blades, which are enclosed in the inner rotating cylinder. We apply the `blockMesh` utility to create the background mesh for the main block, and thereafter apply the `SnappyHexMesh` utility to generate a mesh system for the turbine blades and the rotating cylinder.

3.2.3 BlockMesh

The first step in the `blockMesh` phase is to generate a background mesh of hexahedral cells that fill up the whole domain within the external boundary. Figure 3.5a illustrates the hexahedral control volume where the entire flow is placed. The control volume dimensions have been chosen appropriately to capture the necessary aerodynamic features of the flow. The inlet and outlet patches are faces perpendicular to x–axis and represent the entry and exit domains of the wind into the turbine system. The other patches sideBack and sideFront are perpendicular to the rotational axis (y–axis), and topwall (up) and bottomwall (down) are lower and upper faces in which the rotating domain is enclosed. A sizable background mesh (see Figure 3.5b) is created from the OpenFOAM dictionary file called the **blockMeshDict**, which is found in the `polyMesh` folder of the constant directory. The **blockMeshDict** entries and meshing parameters are given in Appendix A.



(a) Control volume configuration for the Lux geometry with boundary patches



(b) 3D background mesh with blockMesh dictionary

Figure 3.5: 3D geometry by blockMesh

3.2.4 SnappyHexMesh

The OpenFOAM `SnappyHexMesh` process is quite different from the typical approach of doing pre-processing for CFD. The `SnappyHexMesh` utilizes automated algorithms to generate an orthogonal hexahedral mesh system either around or inside a domain of interest. The geometry surfaces are provided in STL format and can be created by a computer aided-design or some other pre-processing packages like SALOME or gmsh [65]. In principle, `SnappyHexMesh` tool aids fast and robust meshing for any complex geometries and by extension supports automated computation procedures.

The meshing process was developed in two steps; the creation of the control volume with the `blockMesh` tool and the surface refinement near the blades with the `SnappyHexMesh` tool. An STL file of the case geometry and a background mesh must be provided to apply `SnappyHexMesh`. The `SnappyHexMesh` tool then performs a 3-stage meshing process of castillation, snapping, and boundary layer refinement. The **SnappyHexMeshDict** dictionary file given in Appendix A consists of six main sections with many meshing parameters to control the behavior of the `SnappyHexMesh`.

The first phase of the SnappyHexMesh is castellation. In this step, the background mesh generated by blockMeshDict is refined based on the surface and volume refinement parameters to create an arbitrary unstructured mesh for the computational domain. The main controls in the castellation iterative procedure are:

- ***maxGlobalCells***: This indicates the overall cell limit on the generated mesh. The castellation process stops when the number of cell (mesh size) specified is reached. In our case, maxGlobalCells is set to

$20,000,000$ cells. Similarly, the *maxLocalCells* indicates the amount of cell limit each processor can refined and stored during the castellation phase.

- ***minRefinementCells***: This indicates the minimum total of cells to be refined during the refinement iteration of the castellation phase. The *minRefinementCells* is defined to avoid over refinement on a limited number of cells and cause a refinement phase to stop if the refined cells exceed the minimum cells.

- ***maxLoadUnbalance***: This defines a control to regulate the maximum imbalance for the processors during parallel meshing process.

- ***nCellsBetweenLevels***: This control regulates the required number of buffer volume cell layers at different stages of the refinement process. The higher the number (though the more refined cells we obtain), the longer time it takes to complete meshing process. So a compromise must be maintained between the meshing time and the quality of mesh refinement.

- ***refinementSurfaces***: This specifies the level of refinement needed to make a good quality mesh on the surface or edge of the input geometry. Sometimes, generating a smooth refinement on the edges of the input geometries is complicated. In this case, two refinement levels can be specified for every STL surface. The first number represents the minimum level, while the second indicates the maximum level of refinement.

- ***resolveFeatureAngle***: This defines the size of the feature angle necessary for the refinement up to the maximum level indicated in the *refinementSurfaces*. The control setting is automatically activated when there is difference between the minimum and maximum surface refinement levels. That is, the maximum level of refinement is enforced on the cells that overlap the edge at an angle exceeding this value.

- ***locationInMesh***: This is used to localize the cartesian points of the region to retain inside the fluid domain during meshing.

After the castellation meshing phase is the surface snapping process. In this step, the patch faces are projected onto the STL surfaces by displacing the vertices in the castellated boundary onto the STL surfaces and solved for the relaxation of the interior mesh with the current displaced boundary vertices. The snapping process is constrained by five parameters that control the number of iterations and tolerance between the generated mesh and STL surfaces.

- ***nSmoothPatch***: This is used to indicate the required number of pre-smoothing iterations along the surface of the patches before projecting on the geometry surface. The larger the value of *nSmoothPatch*, the better the snapping and the smoother the mesh surface. However, the meshing time is longer.

- ***Tolerance***: This indicates the relative distance the snapcontrols algorithm would scan for a point to snap to the surface. That is, the tolerance represents the scale factor of edge length for the points to be attracted onto the surface. The higher the tolerance value, the better and easier it is to find the correspondence.

- ***nSolverIter***: This indicates the total number of iterations for the whole snapping algorithm. That is, the number of interior smoothing iterations applied to snap displacement field. Higher values lead to better mesh quality with equidistant mesh size but longer meshing time.

- ***nRelaxIter***: This defines the required number of relaxing iterations sufficient to eliminate skewed cells or mesh points.

- ***nFeatureSnapIter***: This is used to determine the number of iterations for the snapping iterative process to attract mesh point to the surface in order to prevent coarse edges.

Layer controls

At the closure of the snapping iterative process, the quality of mesh around the blade boundary patch is improved using the `addLayers` functionality. This process can be set in the layer control features. Layer addition is an arbitrary stage in meshing that inserts an extra hexahedral cells along the boundary surface to capture better aerodynamic characteristics on the patch. The addition of layers on the mesh involves shrinking the earlier generated mesh from the boundary surface and inserting the layers of cells through the following steps:

- The initial mesh is cast away from the boundary surfaces by a fixed amount of thickness in the direction perpendicular to the surface.

- The relaxation of the interior mesh is reconstructed according to the current displaced boundary vertices.

- The resulting mesh is iteratively inspected to meet the validation criteria of the algorithm; otherwise, the initial displacement is scaled back by reducing the projected thickness.

- The specified mesh layers are added onto the boundary surfaces given that the validation criteria are met; otherwise, the layering is removed and mesh reversed to the earlier state.

These highlighted steps can be achieved using the `AddLayerControl` feature of the SnappyHexMesh tool. The `AddLayerControl` process starts by stating the number of layers to be added on the specified patches. Its main control parameters are:

- ***relativeSizes***: This is used to indicate the method by which layer parameters particularly the minimum and maximum thickness of the final layer on the boundary surface is defined. The feature is set as either True or False.

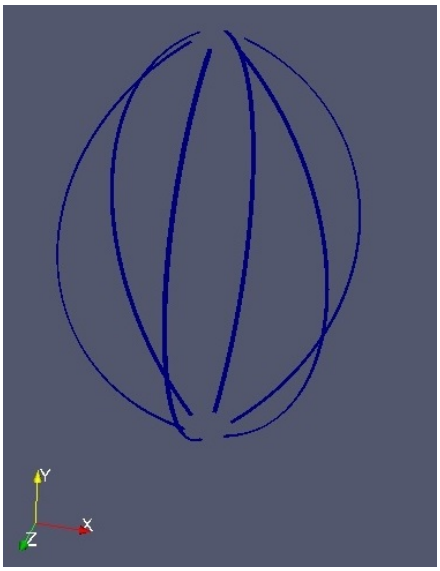
True: When *relativeSizes* is set as true, the layer size parameter is automatically defined in proportion to the size of the volume mesh nearest to the boundary surface. In this regard, *relativeSizes* is assigned the product value of the stipulated cell size and the mesh size outside the boundary surface.
False: Layer criteria are specified precisely by the value in absolute units.

- ***expansionRatio***: This is set to define the sequential growth of the successive layers on the boundary surfaces.
- ***finalLayerThickness***: This defines the height (thickness) of the latest layer relative to the adjacent surface mesh size.
- ***minThickness***: This indicates the overall minimum thickness below which height layers automatically collapse, and must be less than the total thickness of all the boundary layers.
- ***featureAngle***: This is used to set the feature angle to regulate the generation of boundary layers on the surfaces and avoid automatic collapse of the added layers. It is a first-hand parameter to examine if any layer addition fails.
- ***nSmoothSurfaceNormals***: This indicates the number of smoothing iterations to be performed on the surface point standard for the extrusion of the layers.
- ***nSmoothNormal***: This is used to indicate the total number of smoothing iterations sufficient to move the interior mesh.
- ***nSmoothThickness***: This indicates the number of smoothing operations to be performed on the whole layer thickness over several surface patches.
- ***maxFaceThicknessRatio***: This indicates the maximum acceptable value of the aspect ratio for the layer growth generation on the boundary surface especially over highly skewed cells.
- ***maxThicknessToMedialRatio***: This expresses the highest ratio of the layer thickness to the medial distance during the mesh displacement from the boundary surface and adjust the layer growth where the ratio is more than the indicated value.
- ***nLayerIter***: This indicates the total iterations for layer addition algorithm. In addition, it can be used to stop layer addition algorithm instantly after the stipulated value while the mesh still contains some skewed cells.

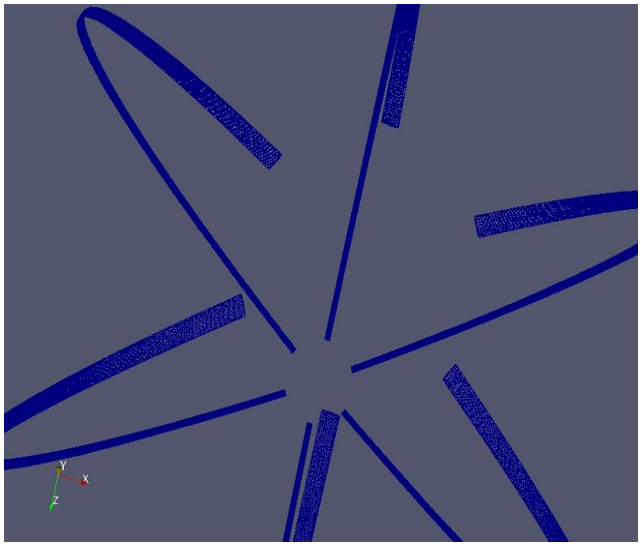
All the iterative procedures in the `SnappyHexMesh` process can be summarized in Table 3.3. After all the meshing factors have been properly defined, the meshing process is automated by running the **SnappyHexMeshDict** in the terminal within the case directory. Figures 3.6 illustrates the generated meshes for the blades and the rotatory region of our geometry using the **SnappyHexMeshDict** utility in OpenFOAM.

Keyword	Description	Examples
castellatedMesh	Generate the castellated mesh?	true
snap	Perform the surface snapping iteration?	true
addLayers	Insert surface layers?	true
mergeTolerance	Combine tolerance as ratio of bounding box of preliminary mesh	1e–06
debug	Regulates writing of transient meshes and screen printing	
	–Save final mesh only	0
	–Save intermediate meshes	1
	–Save `volScalarField` with cellLevel for post–processing	2
	–Save latest intersections as .obj files	4
geometry	Sub-dictionary of all STL surface geometry used	
castellatedMeshControls	Sub-dictionary of controls for castellated mesh	
snapControls	Sub-dictionary of controls for surface snapping	
addLayersControls	Sub-dictionary of controls for layer addition	
meshQualityControls	Sub-dictionary of controls for mesh quality	

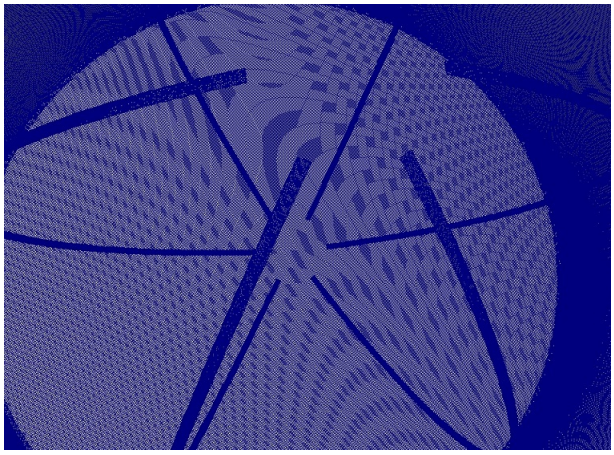
Table 3.3: Summary of SnappyHexMeshDict keywords and description [100].



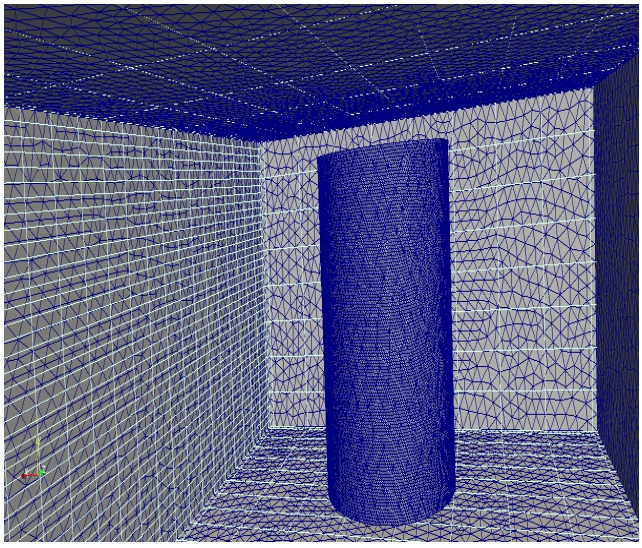
(a) mesh grid on the Lux blades



(b) Lateral view of mesh on the Blades



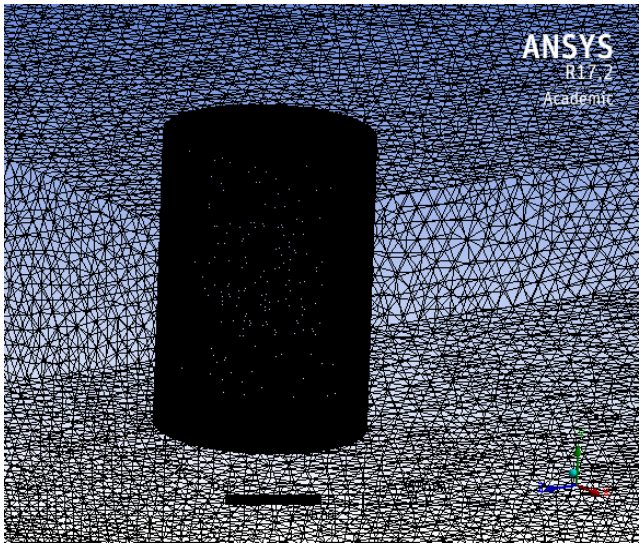
(c) Mesh structure on the inner region of the rotatory zone



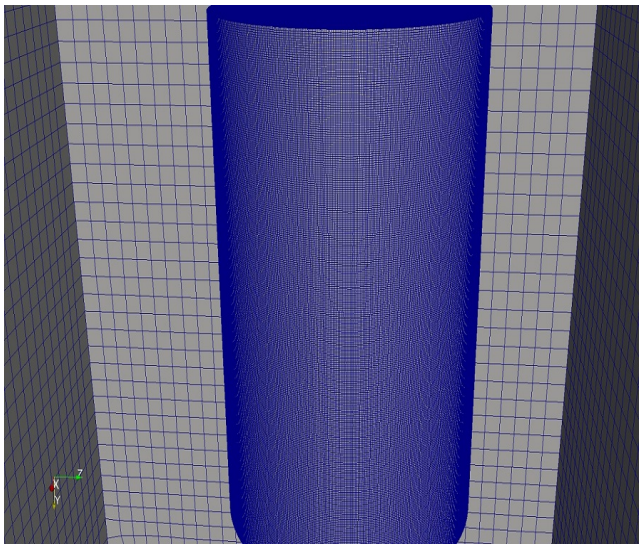
(d) Mesh Grid within the rotatory and fixed domain of the geometry

Figure 3.6: The 3-dimensional mesh of the Lux geometries.

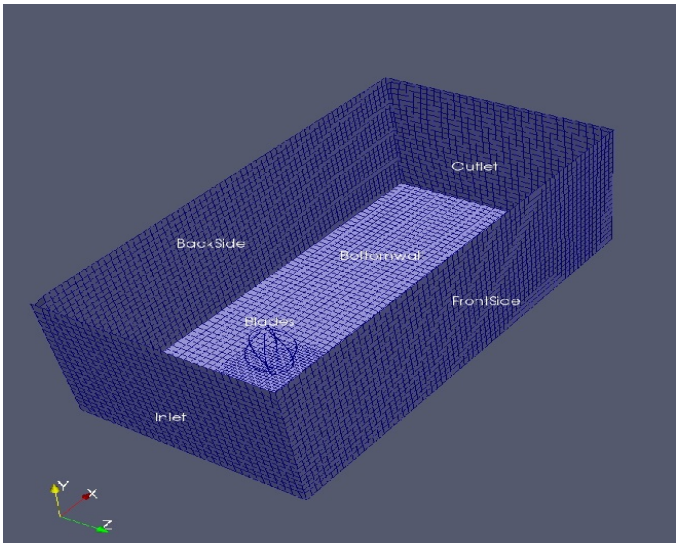
In order to obtain reasonable results comparable to the experimental data, mesh of different sizes and shapes at different level of refinement have been generated and are presented in Figure 3.7.



(a) M1: The first 3D tetrahedral mesh of size 4.9M cells



(b) M2: The second 3D hexahedral mesh of size 7.50M cells



(c) Mesh domain and boundary patches

Figure 3.7: Meshes at different level of refinement.

Figure 3.7 is obtained by changing the control parameters such as the level of surface and region refinement on the blades and the rotatory domain of the turbine geometry. The quality of the mesh obtained via the `snappyhexmeshDict` is examined using the `checkMesh` utility. For all other meshing utilities and features, see OpenFOAM User Guide [100]. Table 3.4 summarizes the characteristic features of the computational meshes, one obtained from the Ansys Mesher and the other with OpenFOAM `snappyhexmeshDict`.

Mesh identifier	Number of cells	Faces per cell	Max skewness	Maximum y^+ on blades	Type of Mesh
Medium M1	4,971,666	4.00	1.92	145.47	Unstructured (Tetrahedral shape)
Fine M2	7,502,065	6.28	1.04	74.88	Unstructured (Hexahedral shape)

Table 3.4: Details of the computational mesh.

3.3 Solver - PimpleDyMFoam

In OpenFOAM, there are many solvers and algorithms to solve different types of flows modeled by the Navier–Stokes equations. Some of these solvers are designed particularly to solve transient or non-transient flows in either inertial or non-inertial reference frames. For instance, Multiple Reference Frame (MRF) solvers such as MRFSimpleFoam solves the flow equations in non-inertial reference frames. However, in most VAWT simulations, the rotor along with the airfoils (rotor blades) is usually set in the rotating reference frame, whereas the inlet and outlet boundaries are defined in the stationary reference frame. So in this case, the MRF solver may no longer tenable to resolve the wake within the stationary domain. Hence there is a need for a rotating (moving mesh) solver, which can also revolve the mesh cells around the rotor and blades while solving the flow equations.

PimpleDyMFoam, a unique moving mesh solver for turbulent incompressible flow is used in this thesis. The PimpleDyMFoam code solves the URANS equations for an incompressible fluids using PIMPLE (Merged PISO-SIMPLE) algorithm procedures [7]. The PIMPLE algorithm combines both the implementation of Semi-Implicit Method of Pressure Linked Equation (SIMPLE) and Pressure Implicit Split Operator (PISO) to solve for the unknowns in the discretized Navier–Stokes equations.

PIMPLE incorporates an iterative method to handle the pressure-velocity coupling of the implicitly discretized flow equations on the dynamic meshes when no over-relaxation factor is given. PIMPLE component PISO contains a robust transient algorithm that allows a reasonably large time-step and takes less computational time to obtain a stable solution. PISO also utilizes a splitting operation for the solution of the discretized momentum and pressure equations [64] on the moving grid system (see Appendix B). A SIMPLE algorithm, on the other hand, handles only a steady-state problem using the pressure-velocity coupling procedures [102] described in section 3.3.2. PIMPLE implementation, as illustrated in Figure 3.8, requires an iterative procedure based on guessing and then correcting the solution to the unknown quantity in the discretized governing equations in order to satisfy mass conservation using predictor-corrector steps.

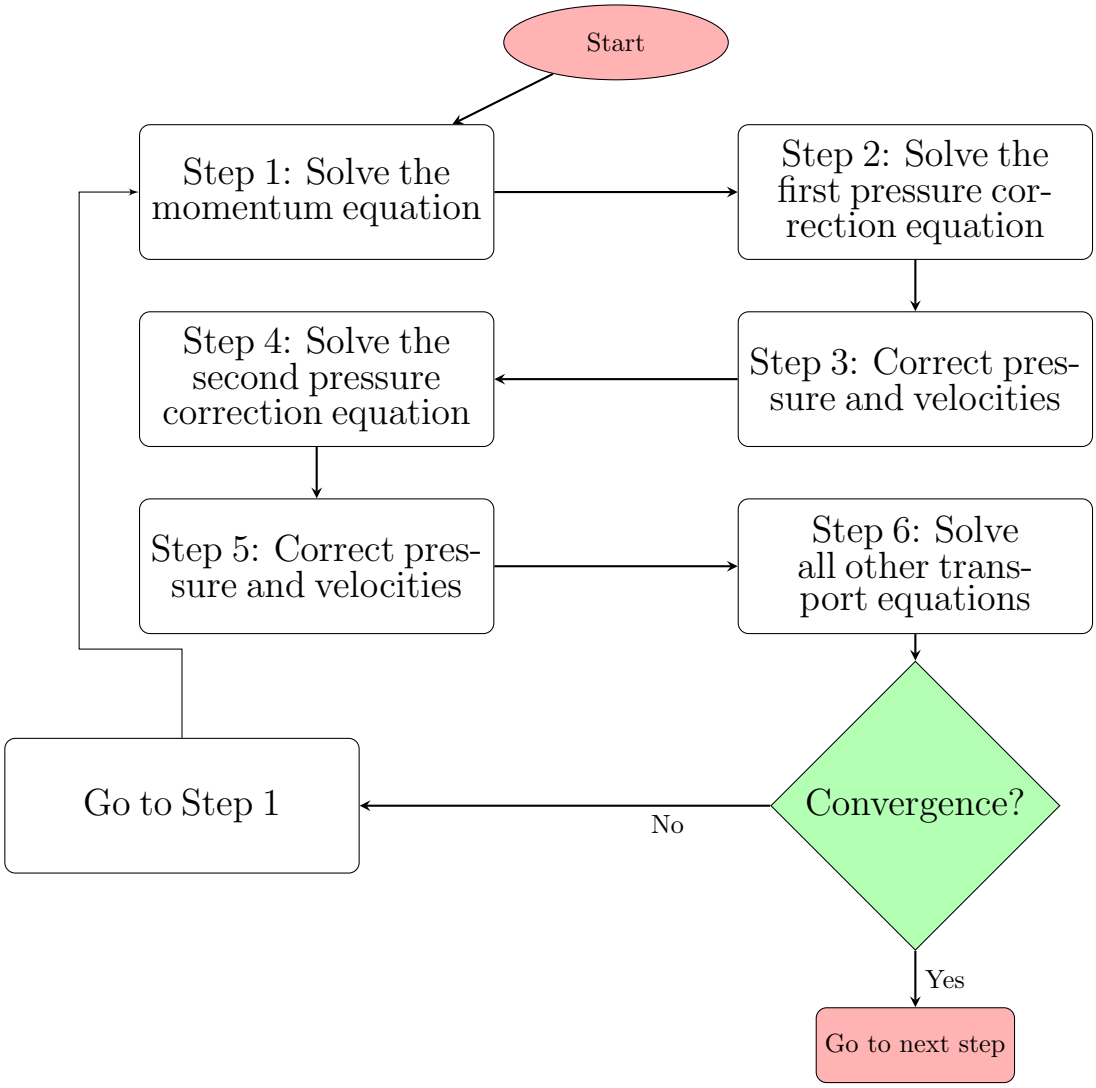


Figure 3.8: PIMPLE flowchart

3.3.1 Governing equations in PimpleDyMFoam

For the Lux VAWT turbulent flow, the URANS equations for the incompressible fluid are given as

$$\nabla \cdot \overline{\mathbf{U}} = 0, \tag{3.1}$$

$$\frac{\partial}{\partial t}\left(\overline{\mathbf{U}}\right) + \nabla \cdot (\overline{\mathbf{U}}\overline{\mathbf{U}}) = \mathbf{g} - \frac{1}{\rho}\nabla \overline{p} + \nabla \cdot (\nu_t \nabla \overline{\mathbf{U}}) - \nabla \cdot (\overline{\mathbf{U}'\mathbf{U}'}), \tag{3.2}$$

or in component form

$$\begin{cases} \nabla \cdot \overline{\mathbf{U}} & = 0, \\ \frac{\partial \overline{U_1}}{\partial t} + \nabla \cdot (\overline{U_1}\overline{\mathbf{U}}) & = -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_1} + \nabla \cdot (\nu_t \nabla \overline{U_1}) - \frac{1}{\rho}\left[\frac{\partial(\rho\overline{U_1'^2})}{\partial x_1} + \frac{\partial(\rho\overline{U_1'U_2'})}{\partial x_2} + \frac{\partial(\rho\overline{U_1'U_3'})}{\partial x_3}\right], \\ \frac{\partial \overline{U_2}}{\partial t} + \nabla \cdot (\overline{U_2}\overline{\mathbf{U}}) & = -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_2} + \nabla \cdot (\nu_t \nabla \overline{U_2}) - \frac{1}{\rho}\left[\frac{\partial(\rho\overline{U_2'U_1'})}{\partial x_1} + \frac{\partial(\rho\overline{U_2'^2})}{\partial x_2} + \frac{\partial(\rho\overline{U_2'U_3'})}{\partial x_3}\right], \\ \frac{\partial \overline{U_3}}{\partial t} + \nabla \cdot (\overline{U_3}\overline{\mathbf{U}}) & = -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_3} + \nabla \cdot (\nu_t \nabla \overline{U_3}) - \frac{1}{\rho}\left[\frac{\partial(\rho\overline{U_1'U_3'})}{\partial x_1} + \frac{\partial(\rho\overline{U_2'U_3'})}{\partial x_2} + \frac{\partial(\rho\overline{U_3'^2})}{\partial x_3}\right], \end{cases} \tag{3.3}$$

with k–ω SST turbulence model of the form

$$\nu_t = \frac{a_1 k}{\max\left(a_1\omega, SF_2\right)},$$

where the turbulence model equations are given in equations (2.36)–(2.38) along with the closure coefficients described in Table 2.2. The three-dimensional URANS equations are nonlinear and require both initial and boundary conditions (2.13)–(2.18) from which the solver can start the solution process.

3.3.2 Pressure-Velocity Coupling

With no transient term in equation (3.1), the flow velocities at the latest time level can be implicitly formulated from the momentum equations using the continuity equation as a constraint to the velocity field. However, the absence of an independent equation for the pressure term poses a challenge for solving the momentum equations. Therefore, a relation defining a field pressure that satisfies the continuity while evaluating the velocities is required.

Following Hrvoje [68], the pressure equation is derived from the URANS system (3.3) around a control volume V_{P} using the semi-discretized momentum equations of the form

$$\alpha_{\mathrm{P}}\overline{\mathbf{U}}_{\mathrm{P}} + \sum_{\mathrm{N}} a_{\mathrm{N}}\overline{\mathbf{U}}_{\mathrm{N}} = \overline{\mathbf{Q}} - \nabla \overline{p}, \tag{3.4}$$

where α_{P} and a_{N} are the diagonal and off-diagonal coefficients of a function $\overline{\mathbf{U}}$, and $\overline{\mathbf{Q}}$ denotes the source vector that incorporates the source component of the temporal term and all other source terms that may be available in the flow (from rotation and turbulent stresses), excluding the pressure gradient term that is set apart to derive the pressure equation. The summation in equation (3.4) is performed over all the control

volumes in the computational domain that share a face with the identified control volume V_{P}. Thus, we introduce a new operator

$$\mathbf{L}(\overline{\mathbf{U}}) = -\sum_{\mathrm{N}} a_{\mathrm{N}} \overline{\mathbf{U}}_{\mathrm{N}} + \overline{\mathbf{Q}},$$

which contains two main parts: the "transport part" together with the matrix coefficients for all neighbors multiplied by the corresponding velocities $-\sum_{\mathrm{N}} a_{\mathrm{N}} \overline{\mathbf{U}}_{\mathrm{N}}$, and the "source part". For example, the $\mathbf{L}(\overline{\mathbf{U}})$ for the incompressible Navier–Stokes equations with the backward Euler temporal differencing is

$$\mathbf{L}(\overline{\mathbf{U}}) = -\sum_{\mathrm{N}} a_{\mathrm{N}} \overline{\mathbf{U}}_{\mathrm{N}} + \frac{\overline{\mathbf{U}}^{n-1}}{\Delta t}. \tag{3.5}$$

The operator $\mathbf{L}(\overline{\mathbf{U}})$ is dependent on both the current velocity $\overline{\mathbf{U}}$ and the velocity of the preceding level $\overline{\mathbf{U}}^{n-1}$ and sometimes the two previous time level $\overline{\mathbf{U}}^{n-2}$ when the BDF2 temporal differencing is applied. However, the matrix coefficient a_{N} in equation (3.5) relies on the face mass fluxes, which is constructed from the velocity field, thereby resulting in a non-linear system.

If we resolve equation (3.4) following the Rhie–Chow interpolation [37], the flow velocity at the cell center is derived as

$$\overline{\mathbf{U}}_{\mathrm{P}} = \frac{\mathbf{L}(\overline{\mathbf{U}})}{\alpha_{\mathrm{P}}} - \frac{1}{\alpha_{\mathrm{P}}} \nabla \overline{p},$$

while the velocity at the cell face (f) is approximated via interpolation from the cell center values to get

$$\overline{\mathbf{U}}_f = \left(\frac{\mathbf{L}(\overline{\mathbf{U}})}{\alpha_{\mathrm{P}}} \right)_f - \left(\frac{1}{\alpha_{\mathrm{P}}} \right)_f (\nabla \overline{p})_f \,. \tag{3.6}$$

This equation is applied afterward to evaluate the face fluxes. In the meantime, the discretized continuity equation can be expressed as

$$\nabla \cdot \overline{\mathbf{U}} = \sum_f \mathbf{S} \cdot \overline{\mathbf{U}}_f = 0, \tag{3.7}$$

where $\mathbf{S}$ is the outward-pointing face area vector and $\overline{\mathbf{U}}_f$ is the velocity vector value at the cell face. Substituting equation (3.6) into equation (3.7) results in the discretized pressure equation

$$\nabla \cdot \left(\frac{1}{\alpha_{\mathrm{P}}} \nabla \overline{p} \right) = \sum_f \mathbf{S} \cdot \left[\left(\frac{1}{\alpha_{\mathrm{P}}} \right)_f (\nabla \overline{p})_f \right] = \sum_f \mathbf{S} \cdot \left(\frac{\mathbf{L}(\overline{\mathbf{U}})}{\alpha_{\mathrm{P}}} \right)_f .$$

Therefore, the complete form of the discretized incompressible Navier–Stokes system is

$$\alpha_{\mathrm{P}} \overline{\mathbf{U}}_{\mathrm{P}} = \mathbf{L}(\overline{\mathbf{U}}) - \nabla \overline{p}, \tag{3.8}$$

$$\sum_f \mathbf{S} \cdot \left[\left(\frac{1}{\alpha_{\mathrm{P}}} \right)_f (\nabla \overline{p})_f \right] = \sum_f \mathbf{S} \cdot \left(\frac{\mathbf{L}(\overline{\mathbf{U}})}{\alpha_{\mathrm{P}}} \right)_f , \tag{3.9}$$

while the face mass fluxes F are evaluated using $\overline{\mathbf{U}}_f$ from equation (3.6):

$$F = \mathbf{S} \cdot \overline{\mathbf{U}}_f = \mathbf{S} \cdot \left[\left(\frac{1}{\alpha_{\mathrm{P}}} \right)_f (\nabla \overline{p})_f \right]. \tag{3.10}$$

Evidently, when equation (3.7) is satisfied, the conservation of the face flux is guaranteed.

The discretized Navier–Stokes system (3.8)–(3.9) is coupled in velocity and pressure. The equations show a linear dependence of velocity on pressure and vice-versa. Thus, finding an explicit solution to the unknowns requires a series of iterations from the classical iterative algorithms or sequential algorithms [68]. In CFD, however, the selection of the solution algorithms (or solvers) is informed by the type of the grid structure, the size of the computational nodes, and most importantly, the structure of the coefficient matrices of the equations being solved. The iterative algorithms [128] work by solving the whole discretized system of equations concurrently over the entire computational domain. If the algebraic equations are non-linear, the iterative algorithm applies first the initial solutions to linearize the given equations, and then systemically improves the approximate solutions until converged solutions are obtained. Such an iterative procedure is most beneficial when the number of mesh points and the size of the system of equations are relatively small. However, due to the high cost of computation with the direct and iterative algorithms, the frequently used algorithms are the sequential algorithms such as PISO and SIMPLE [103].

The sequential algorithms, also called the uncoupled solvers in CFD, implement pressure-velocity coupling procedure in OpenFOAM [103]. A sequential algorithm operates by solving the three momentum equations sequentially before applying the updated velocity field to estimate the pressure equation for continuity. That is, the basic idea behind uncoupled solver is to solve each discretized equation separately, and because the flow equations are usually non-linear, and sometimes coupled, multiple iterations of the solution algorithm must be performed until a converged solution is obtained. A typical example of the uncoupled solver is the PimpleDyMFoam solver, which implement the PIMPLE algorithm [64] to solve the discretized Navier–Stokes equations. As a fluid flow algorithm, SIMPLE allows to couple the discretized Navier–Stokes equations in the following iterative steps:

Note: n and i represent the time step and iteration counter respectively.

1. Increment time :

$$t^n = t^{n-1} + \Delta t.$$

2. Initialize SIMPLE inner iteration with :

$$\overline{\mathbf{U}}^{n,0} = \overline{\mathbf{U}}^{n-1},$$

$$\overline{\mathbf{p}}^{n,0} = \overline{\mathbf{p}}^{n-1},$$

$$\mathbf{L}^{n,0} = \mathbf{L}^{n-1}.$$

- Apply the initial conditions to solve the momentum predictor equation

$$\frac{\alpha_{\mathrm{P}}}{\alpha_U}\overline{\mathbf{U}}_{\mathrm{P}}^{n,i} = \mathbf{L}_n^{i-1}(\overline{\mathbf{U}}^{n,i}, \overline{\mathbf{U}}^{n-1}) - \sum_f \mathbf{S}\left(\overline{p}^{n,i-1}\right)_f + \frac{1-\alpha_U}{\alpha_U}\alpha_{\mathrm{P}}\overline{\mathbf{U}}_{\mathrm{P}}^{n-1}, \tag{3.11}$$

for the intermediate velocity field $\overline{\mathbf{U}}^{n,i}$ with the available face fluxes, while the pressure gradient term is estimated by the pressure distribution from the initial guess. The equation is under-relaxed [46] implicitly with the velocity under-relaxation parameter α_U $(0 < \alpha_U \leq 1)$.

$$\sum_f \mathbf{S} \cdot \left[\left(\frac{1}{a_{\mathrm{P}}}\right)_f \left(\nabla \overline{p}^{n,i}\right)_f\right] = \sum_f \mathbf{S} \cdot \left(\frac{\mathbf{L}_n^{i-1}(\overline{\mathbf{U}}^{n,i}, \overline{\mathbf{U}}^{n-1})}{\alpha_{\mathrm{P}}}\right)_f. \tag{3.12}$$

- Formulate and solve the pressure equation (3.12) to find the new pressure field $\overline{p}^{n,i}$.
- Evaluate the latest set of conservative face fluxes satisfying the continuity equation by

$$F = \mathbf{S} \cdot \overline{\mathbf{U}}_f = \mathbf{S} \cdot \left[\left(\frac{\mathbf{L}_n^i(\overline{\mathbf{U}}^{n,i}, \overline{\mathbf{U}}^{n-1})}{\alpha_{\mathrm{P}}}\right)_f - \left(\frac{1}{\alpha_{\mathrm{P}}}\right)_f \left(\nabla \overline{p}^{n,i}\right)_f\right]. \tag{3.13}$$

  The operator $\mathbf{L}$ is reconstructed using the new set of conservative face fluxes - $\mathbf{L}_n^i$.

- Adjust the pressure field solution with the pressure under-relaxation parameter $\alpha_{\overline{p}}$ $(0 < \alpha_{\overline{p}} \leq 1)$.

$$\overline{p}^{n,i} = \overline{p}^{n,i-1} + \alpha_{\overline{p}}\left(\overline{p}^{n,i} - \overline{p}^{n,i-1}\right). \tag{3.14}$$

- Adjust the velocity field correctly using

$$\overline{\mathbf{U}}_{\mathrm{P}}^{n,i} = \left(\frac{\mathbf{L}_n^i(\overline{\mathbf{U}}^{n,i}, \overline{\mathbf{U}}^{n-1})}{\alpha_{\mathrm{P}}}\right) - \left(\frac{\nabla \overline{p}^{n,i}}{\alpha_{\mathrm{P}}}\right)_f, \tag{3.15}$$

  with the newly constructed face fluxes and pressure field.

- Solve every other equations in the system (such as the turbulent kinetic energy equation), using the newly constructed conservative fluxes, pressure and velocity fields.
- Run the inner iteration until the acceptable tolerance for the pressure-velocity system is reached. If the solution fails to converge, repeat the inner iteration with $i = i + 1$ from the first step in 2. Otherwise, start a new time step from 1 until the stipulated time is reached.

Similarly, the PISO algorithm for transient flow analyses [64] is described as follows:

1. State all the initial conditions for the flow fields such as initial pressure $\overline{p}^*$, initial velocity $\overline{\mathbf{U}}^*$ and their corresponding boundary conditions.

2. Evaluate the new time step value for the calculation using the initial time step size Δt and $t^n = t^{n-1} + \Delta t$.

3. Solve the discretized momentum equations (3.8) – (3.9) to estimate an intermediate velocity field. Although the pressure gradient source term is not exactly known at this stage, the pressure field value at the preceding time-step is applied instead.

4. Evaluate equation (3.13) for the conservative mass fluxes at the cell faces.

5. Solve for the pressure equation (3.14) and impose the corresponding under-relaxation parameter.

6. Apply correction to the mass fluxes at the cell faces following equation (3.11) in case the continuity equation is not satisfied. Use pressure correction factor $\overline{p'}$ and estimate the velocity correction $\overline{\mathbf{U}'}$ and the corrected pressure and velocity fields by

$$\overline{p} = \overline{p'} + \overline{p}^*, \, \overline{\mathbf{U}'} = D\left(\overline{p'}^n - \overline{p'}^{n-1}\right), \, D = \left(\frac{\mathbf{A}}{\alpha_{\mathrm{P}}}\right), \, \overline{\mathbf{U}} = \overline{\mathbf{U}'} + \overline{\mathbf{U}}^*.$$

7. Update the boundary condition based on the new velocities and pressure fields.

8. Solve every other equations in the system (such as the turbulent kinetic energy equations) and calculate the eddy viscosity from the turbulence equations using the newly constructed conservative fluxes, pressure and velocity fields.

9. If final time is not reached, repeat from 3.

10. Increase the time step and go over step 1 until a prescribed tolerance is obtained.

Compared to the SIMPLE algorithm, step 5 and 6 can be iterated for a stipulated number of times to correct the solution in the case of highly non-orthogonal mesh.

3.3.3 Discretization Schemes

OpenFOAM offers a flexible choice of discretization and interpolation schemes between computational points. The discretization schemes applied to each of the terms in the discretized URANS equations (3.8) – (3.9) are given in Table 3.5. The chosen schemes were based on the recommendations from the OpenFOAM guide and tutorial cases for the RANS simulation [100] and were first- and second-order accurate. However, the discretization schemes can be of any order in principle.

These discretization schemes and other algorithms can be set in OpenFOAM using the *fvScheme* dictionary given in Appendix B, which contains the settings for the matrix solvers, algorithm controls, preconditioners, discretization schemes, and tolerances required to solve the system equations. Additionally, evaluating the pressure equation has been observed to be the most computational demanding with nearly 80 percent of the simulation time spent on re-evaluating the pressure equation in order to satisfy mass conservation. The geometric-algebraic multi-grid (GAMG) solver [82] has been adopted in the *fvSolution* dictionary to proffer solution to the pressure equation. GAMG is considered a robust solution method with a relatively quick solution procedure that accelerates the convergence of the iterative process [90]. GAMG operates on

Term	Numerical scheme
DDtScheme	BDF2, CrankNicolson, TR-BDF2
Grad	linear scheme (see equation (2.59))
Div(ϕ, U)	linear upwind scheme (see equation (2.62))
div(ϕ, k)	Upwind scheme (see equation (2.60))
Div(ϕ, ω)	Upwind scheme
Div(ϕ, ϵ)	Upwind scheme
Div($\nu_{eff} * (T(grad(U)))$	linear scheme
Laplacian	limited linear differencing scheme (see equation (2.63))
Interpolation	linear scheme
SNGrad	limited linear differencing scheme

Table 3.5: Discretization schemes used for the simulation of the Lux VAWT.

the basic principle of utilizing a coarse mesh with a quick solution to smooth out high-frequency errors and compute an initial solution for the finer mesh [13]. It also applies many levels of progressively coarser discretizations to obtain quick convergence of the iterative method [72]. This is achieved through a geometric coarsening of the mesh domain (geometric multi-grid), or by using the same principle wholly on the matrix coefficient of the fast solution, irrespective of the mesh geometry [117]. When GAMG is applied in practice, the user prescribes the initial mesh system and sets the necessary parameters to coarsen the mesh in stages, and the coarsening is then executed through the *faceAreaPair* (geometric) or algebraic pair setting.

The moving mesh procedure of the GAMG solver relies upon the inputs from the `dynamicMeshDict` file, which contains the information about the moving and static patches. In addition, the solver also handles the mesh motion based on the type of the specified interface. The sliding mesh interface technique [136] is used to handle the rotating mesh within the computational domain. In practice, the mesh manipulation or interpolation between the rotating mesh and the static counterpart is done essentially seamlessly after each time step. In this way, part of the mesh surface is rotated but not deformed in any way.

DynamicFvMesh

A dynamic mesh such as rotor (or blade) mesh is a mesh with moving boundaries. The rotational motion of the turbine blades is a major issue requiring a dynamic mesh. This is handled by the libraries with the capability of dynamic meshes, which are available in the OpenFOAM dictionary. For example, (multi)SolidBodyMotionFvMesh is a sliding mesh library used to move the whole mesh or parts of the mesh as a solid body during the simulation. SolidBodyMotionFvMesh offers greater flexibility especially for the transient simulation of the VAWT with rotatory components. The attributes of the rotating components such as the center of rotation and the angular velocity of the Lux VAWT are described in the `dynamicMeshDict` file of the constant directory. The `dynamicMeshDict` entries for the Lux VAWT simulation case are given

in Appendix B.

The Boundary patches

For a realistic simulation, it is essential to describe the set of appropriate boundary patches and faces for the Lux VAWT geometry (see Appendix B). When the meshing process is completed the boundary patches such as `Inlet`, `Outlet`, `Bladewalls`, `Bottom-wall`, `Top-wall`, `Side-walls`, and `AMIs` are generated and used as setting for the initial and boundary conditions of the flow. Boundary patches not only serve as geometric entities but also constitute an integral part of the computation and numerics of the boundary conditions. To apply the boundary conditions, the computational boundary is first separated into a set of patches (i.e., regions). These patches can be of a different type with specific attributes presented in Figure 3.9. The base type describes the geometric boundary patch and is contained in the constant/polyMesh directory. However, the primitive and derived types are numerical patches that describe the numerical conditions peculiar to a given physical transport (such as velocity, pressure, etc.) on the patch. They can be found in the field file (0-directory), where the boundary and initial conditions are stated. Only a few of the patches specific to the Lux VAWT simulations are described below.

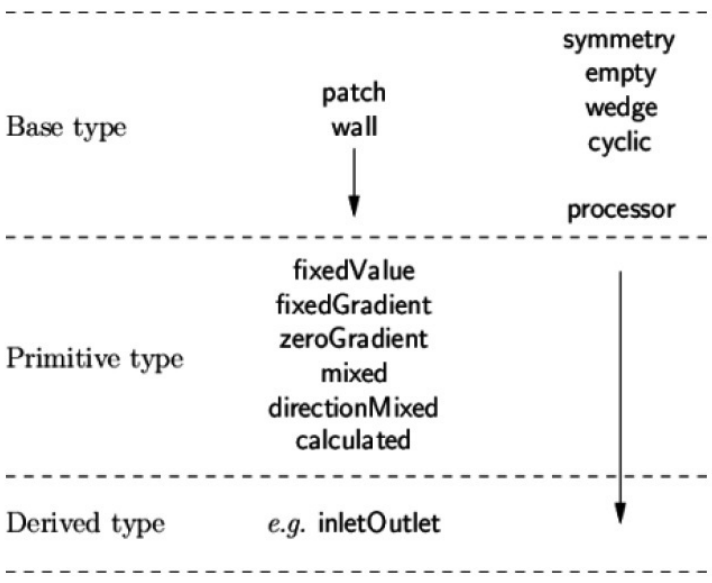


Figure 3.9: Boundary patch types in OpenFOAM [100].

Base Types

- Inlet and Outlet: These patches require no geometric or topological information of the simulated geometry. However, they are applied to prescribe the velocity vector, pressure field, and other scalar properties of the flow at the inlet and outlet boundaries.
- Wall: This patch type is particularly preferred when a computational boundary coincides with a wall. This is essentially applicable in turbulent modeling, where wall distance is a requisite to estimate

turbulent effects.

- Empty: This patch type is required especially for the simulation of one- and two-dimensional flow problems with OpenFOAM. With an empty patch, one can easily generate a three-dimensional geometry of a two-dimensional flow problem by requiring an empty condition on each patch whose plane is perpendicular to the third dimension.
- Symmetry: This is applied to any (non-planar) computational patch identified with the symmetry plane (slip) condition. It requires no input; however, the flow field and geometry must be symmetric. Symmetry patches are typically applied to simulate zero-shear slip walls in viscous flows problem.
- Cyclic: This patch supports the arrangement of two computational patches just as though they are physically connected. One cyclic patch can be connected to another using a `neighbourPatch` keyword in the boundary file.
- CyclicAMI: Like cyclic, this is used to connect two rotating patches whose faces are unmatched. It is commonly used to characterize the sliding interfaces of the rotating domain.

Primitive Types

- fixedValue: By fixed value condition, the value of the transport quantity (such as velocity, pressure, turbulent intensity, etc.) is stated on a certain boundary of the computational domain. However, the assigned value can be constant (fixed value) or variable dependent.
- zeroGradient: Here, the normal gradient of the transport field is set to zero.
- calculated: The transport field value is extrapolated from other fields within the flow domain.
- freestream: The value of the transport field assumes a free-stream condition of the flow. By this condition, the boundary patch assumes a value of the flow condition far off the obstacle. There are however two approaches to implement this boundary condition. When the flow is moving away from domain, a zero gradient can be prescribed. Otherwise, a constant value is allotted.

Derived Types

- inletOutlet: The value of the velocity field or pressure field interchanges between the fixedValue and zeroGradient conditions according to the flow direction.
- movingWall: This boundary condition is used to prescribe the transport field behavior at the moving walls, e.g., moving mesh (AMI) cases. In contrast to the fixed value boundary condition, the transport field value is set relative to the boundary to which it is applied.

In summary, the list of the useful boundary conditions applied in this thesis is given in Table 3.6. For the comprehensive catalog of the boundary conditions implemented in OpenFOAM, see [100]. An example of a velocity field condition at the inlet boundary patch of the flow is illustrated as:

```
inlet
  {
     type    fixedValue;
     value   uniform(0 0 -12)
  }
```

The boundary patch is identified as inlet (i.e., base type) and of type fixedValue with numeric values (0 0 -12), indicating that the inlet velocity of the wind is 12 m/s in the negative x-direction.

Boundary	**Flow Quantities**				
Patches	**U**	**p**	ω	ν_t	k
`Inlet`	fixedValue i.e. Dirichlet uniform (0 0 -8)	zeroGradient (Neumann) i.e., absence of gradient	fixedValue i.e., Dirichlet uniform 3244.95 1/s	calculated from k and ω	fixedValue i.e., Dirichlet uniform 0.24 $m^2 s^{-2}$
`Outlet`	inletOutlet (Neumann) with inlet value (0 0 0)	fixedValue i.e., Dirichlet p_∞	inletOutlet i.e., Neumann uniform 3244.95 1/s	calculated from k and ω	inletOutlet
`Bladewalls`	movingWallVelocity i.e., Dirichlet uniform (0 0 0)	zeroGradient i.e., Neumann	Wall function that OpenFOAM calls omegaWallFunction	nutWallFunction	kqRWallFunction
`Bottom-wall`	fixedValue i.e., Dirichlet uniform (0 0 0)	zeroGradient	omegaWallFunction	nutWallFunction	kqRWallFunction
`Top-wall`	symmetry (no–slip)	symmetry (no–slip)	symmetry (no–slip)	symmetry (no–slip)	symmetry (no–slip)
`Side-walls`	symmetry (no–slip)	symmetry (no–slip)	symmetry (no–slip)	symmetry (no–slip)	symmetry (no–slip)
`AMI1-AMI2`	cyclicAMI with internal value of uniform (0 0 -8)	cyclicAMI with internal value of uniform 0	cyclicAMI with internal value of uniform 3244.95	cyclicAMI with internal value of uniform 0	cyclicAMI with internal value of uniform 0.24

Table 3.6: Boundary conditions for the Lux VAWT simulation case in OpenFOAM.

The inlet boundary patch of the Lux VAWT computational domain is defined with a fixed value of the wind velocity ranging from 4 m/s – 22 m/s sufficient to match the CFD results of the simulations with the experimental data. Similarly, the outlet boundary patch of the computational domain is defined as a constant pressure value. The inlet pressure was assigned a fixed value at the inlet patch. The rest of the patches within the computational domain were defined according to the attributes set in Table 3.6. For example, the `Top-wall` patch was defined as symmetry boundary condition so as to set the normal gradients of all the flow variables to zero and reduce the extent of the computational domain to a symmetric subsection of the overall physical geometry. The `movingWallVelocity` boundary condition is applied to the `Bladewalls` patch with a uniform velocity value of (0, 0, 0). For the turbulent intensity variables k and ω, the inlet values need to be derived. According to Zamani et al. [135], the mixing length in the omega equation can be expressed as

$$l = 0.07L,$$

where L denotes the chord length of the Lux VAWT blades given as 0.2 m. For example, for the case of inlet velocity 4 m/s, along with the turbulence intensity (T_u) of 5% [129], the turbulent kinetic energy is

calculated as

$$k = \frac{3}{2}\left(U_{inlet} T_u\right)^2 .$$

With this value, the inlet value for the ω in the boundary conditions is calculated with

$$\omega = \frac{\sqrt{k}}{C_\mu^{1/4} l}.$$

The aerodynamic parameters used in the calculations are summarized in Table (3.7).

Airfoil	NACA 0012
Simulation type	Unsteady simulation
Wind Speed	4 $\mathrm{m\,s^{-1}}$ – 22 $\mathrm{m\,s^{-1}}$
Tip Speed ratio	1.74 – 7.19
Density	1.225 $\mathrm{kg\,m^{-3}}$
Reynolds number	8×10^5
Pressure	101325 Pa
Fluid	Air
Kinematic Viscosity	1.47922×10^{-5} $\mathrm{m^2\,s^{-1}}$
Empirical constant C_μ	0.09
Turbulence quantities	$k = 6.00 \times 10^{-2}$ $\mathrm{m^2\,s^{-2}}$, $\omega = 17.49$ $\mathrm{s^{-1}}$, $\epsilon = 9.44810^{-2}$ $\mathrm{m^2\,s^{-3}}$
CFD algorithm	PIMPLE
Turbulent model	$k-\omega$ SST

Table 3.7: Computational parameters for the Lux VAWT.

3.4 Post–processing

The basic post-processing utility used by OpenFOAM is ParaFoam. ParaFoam as a built-in module in OpenFOAM uses a software package called ParaView, which is an open-source, multi-platform data analysis and visualization application [67]. The OpenFOAM output, which is usually in FOAM format, is exported to ParaView with the support of ParaFoam. Thereafter, the imported data can be visualized and analyzed interactively in three dimensions with the help of ParaView's batch processing features. For the simulated cases, the post-processing was approached in two steps. The first basic step requires the use of text-based sampling tool of OpenFOAM to isolate data points for additional processing. The second approach applied three-dimensional visualization tools to inspect the flow for unusual behaviors and regular monitoring of the output. For the three-dimensional visualization tool, ParaView is picked because it connects smoothly to the FOAM-cases, whereas some professional commercial tools like GLview Inova needed conversion [41]. The panels of ParaView for our simulation case are shown in Figure 3.10.

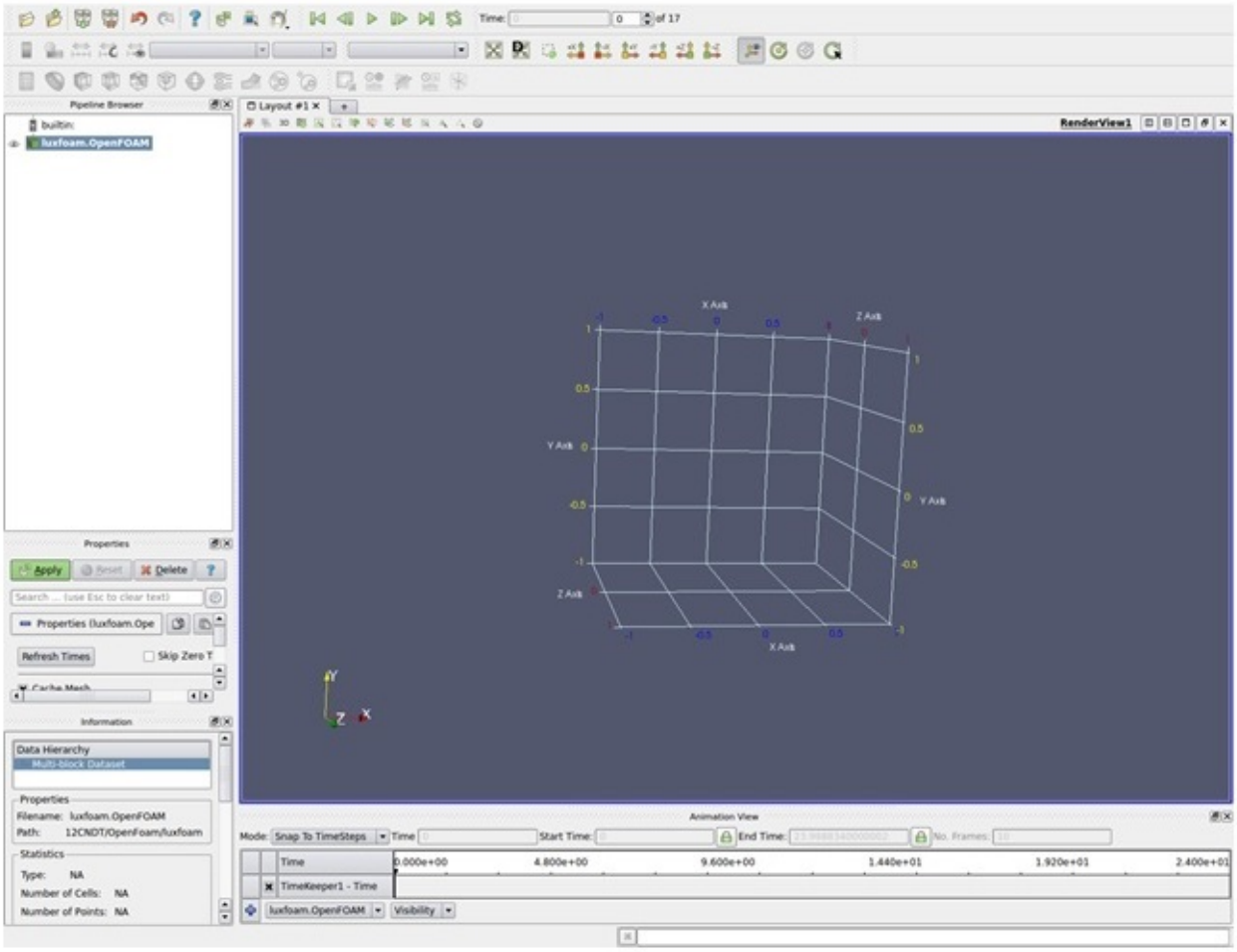


Figure 3.10: Graphical interface of ParaView

4 Simulation Results and Discussion

In this chapter, the numerical simulations of the air flow through the Lux VAWT are completed with the help of the OpenFOAM package. CFD is used to predict the air flow pattern and the power output of the Lux VAWT. The numerical simulations are performed using the concept of sliding mesh technique and MRF capability to investigate the turbine performance. The formulation partitions the computational domain into two sub-domains, namely, the stationary and rotating domains. For the domain surrounding the stator, a stationary reference frame is applied, while the blades and rotor are confined to the rotating reference frame. The cell size of the mesh is relatively the same on both sides of the sliding interface between the rotating and stationary domains in order to attenuate the numerical error at the interface. The characteristic power curves are then used to estimate the operating velocities, power coefficient, and the TSR of the Lux VAWT. The results are presented in both graphical and tabular forms to illustrate possible trends of the aerodynamic characteristics.

4.1 Numerical Simulation

Numerical simulations were performed using an incompressible, pressure-based solver called PimpleDyMFoam in OpenFOAM. The main characteristic parameters of the simulations are given in Table 4.1. The fluid (air) is considered an incompressible with constant density, and considering the high Reynolds number for the turbulent case, the turbulence within the flow regime was modeled with the k–ω SST model in an effort to optimize the performance characteristics of the Lux VAWT that depend on the development of the boundary layer on the turbine blades. Standard wall functions are used for the near-wall treatment of the boundary sub-layers. The simulation model is set up with a second-order upwind discretization scheme and a default under-relaxation parameter (see Appendix B). This allows us to reduce the convective Courant number restriction on the time-stepping schemes. However, the sliding mesh technique adopted during the simulation requires a conservative time-step that is set at $\Delta t = 0.00277$ s for all the simulation cases. This conservative time-step enables the computations to achieve convergence.

The airflow was initially set to be at the atmospheric pressure and a uniform inlet velocity in the computational domain. At the inlet boundary patch of the computational domain, a uniform velocity with a turbulence intensity of 5% was prescribed as the wind free–stream velocity towards the Lux VAWT. This presumption is based on the experimental study by Veers and Winterstein [129], where a turbulence intensity of approximately 5% was realized at a selected wind speed. The turbulent viscosity ratio of about 0.8 is

prescribed to account for a high turbulence viscosity. Within the rotating domain, the normal gradient of the gauge pressure is considered to be zero. However, on the blade surfaces and the computational walls, a no-slip condition was imposed ([24], [108]). All the turbine blades were set up as rigid walls in the rotating domain, subject to the usual no-slip condition. An inlet velocity range of 4 m/s – 22 m/s was used to simulate the flow with respect to the change in the angular velocity of the turbine from 3.124 rad/s to 4.18879 rad/s. During the computation at each grid point, the absolute residual at point `P`

$$R_{\mathtt{P}} = \left| \alpha_{\mathtt{P}} \phi_{\mathtt{P}} - \sum_{nb} a_{nb} \phi_{nb} - \overline{\mathbf{Q}} \right| \tag{4.1}$$

that measures the local imbalance of a conserved transport quantity in each control volume is calculated for each of the conservative governing equation (2.25) and the turbulence model equations (2.36) – (2.37), with nb denoting cell neighbors of cell `P`, and ϕ represents the conserved variable solution for the given equation at a specific time-step in a given iteration loop. At each grid point, the absolute residuals are computed and normalized relative to the local values of the transport quantity so as to quantify the relative error. During each time-step, the simulation is designated to iterate a limited number of times specified in the `fvScheme` file until all the absolute residuals on the turbine blade walls decline below the predefined convergence criterion, at which point a reasonable solution is assumed to be achieved (see Figure 4.1). In all the simulation cases, the minimum convergence criterion was set to 10^{-4} for the continuity, momentum, and turbulent viscosity transport equations. This convergence criterion is assumed to be sufficient for the turbulent flow simulations ([8], [100]). Figure 4.1 illustrates a representative plot of the convergence history of the CFD solution against the runtime of the simulation. At convergence, the numerical computations no longer change appreciably with additional simulation time.

Reynolds number [-]	$133,000$ – $800,000$
Cut in wind speed	4 m/s
Cut out wind speed	25 m/s
Number of cells	4.9 M, 7.5 M
Rotational speed	3.124 rad/s – 4.18879 rad/s
Tip speed ratio (λ)	1.74 – 7.19
Time-step (Δt)	2.777×10^{-3} s
Density (ρ)	1.225 kg m^{-3}
Turbulent model	k–ω SST

Table 4.1: Main simulation parameters

Simulations were consecutively performed until they attained a quasi-steady with a periodic solution. The corresponding physical time required to reach steady state was captured as time spent to complete 16 turbine revolutions and would vary in each case, according to the tip speed ratio and time-stepping schemes. Several

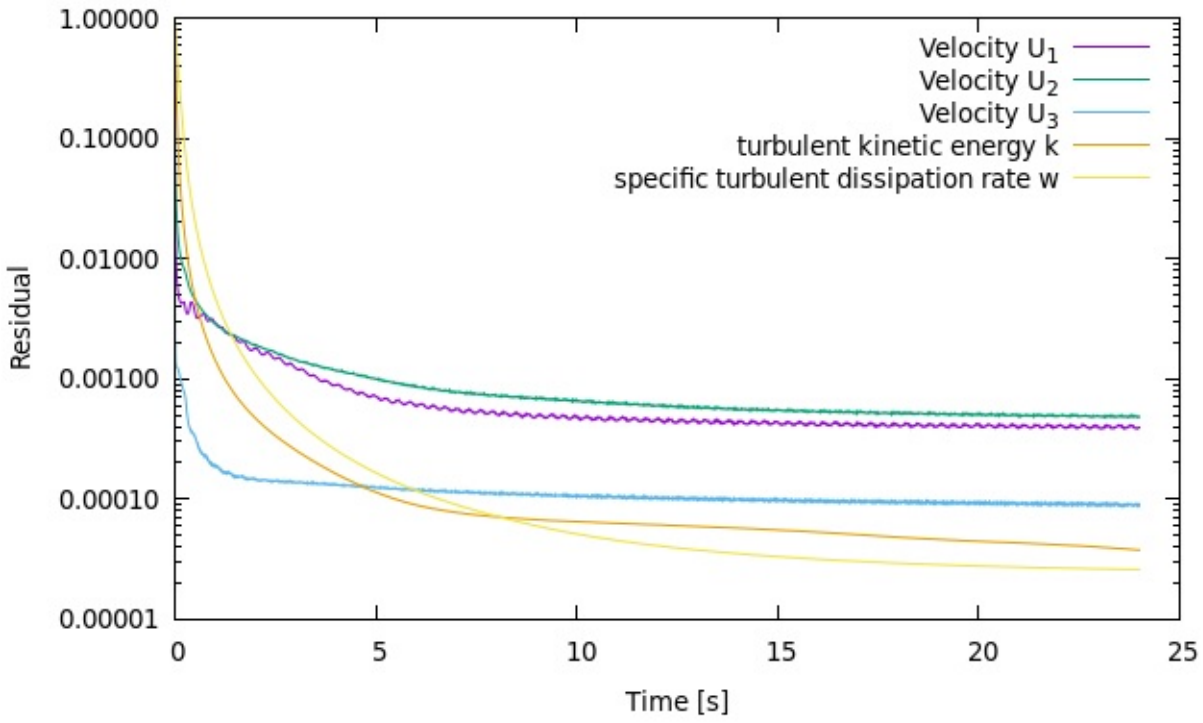


Figure 4.1: Convergence history of the CFD solution at the inlet velocity of 5 m/s. The plot suggests that changes in the transport quantities mostly became negligible after 12 s of the simulation time.

changes were implemented to reduce the simulation time, including changing the discretization schemes and settings for the solvers and pressure-velocity coupling. However, the central parameter limiting the simulation speed is the convective Courant number that greatly affects the explicit numerical schemes. Table 4.2 illustrates the computational runtime for mesh M1 (see Figure 3.7a) at different inlet velocities with different time-stepping schemes.

Vel. ($\mathrm{m\,s^{-1}}$)	BDF2	CN	mTR–BDF2
4	2 : 12 : 53 : 20	4 : 09 : 58 : 49	
5	3 : 20 : 11 : 37	4 : 09 : 00 : 10	
6	3 : 20 : 31 : 21	2 : 12 : 20 : 29	8 : 21 : 05 : 10
7	4 : 05 : 16 : 02	4 : 23 : 47 : 32	
8	3 : 21 : 39 : 14	3 : 22 : 22 : 51	3 : 22 : 06 : 10
9	3 : 14 : 08 : 55	3 : 15 : 26 : 55	
10	3 : 06 : 50 : 20	3 : 15 : 43 : 20	3 : 17 : 39 : 40
11	3 : 03 : 27 : 22	3 : 07 : 21 : 46	
12	1 : 22 : 17 : 36	1 : 20 : 17 : 00	3 : 07 : 10 : 03
13	1 : 22 : 09 : 20	2 : 23 : 23 : 34	

Table 4.2: CPU time [D:H:M:S] $\Delta t = 0.00277$ s (8 proc, 8 GB RAM).

From the table, it can be inferred that the BDF2 gives the least runtime to achieve a steady state solution in most of the simulation cases.

For an accurate estimation of the unsteady term of the turbulent flow, the CFD results of the three time-stepping schemes are compared at a constant time–step $\Delta t = 0.00277$s, i.e., BDF2, CN, and modified TR–BDF2. The performance of each of the three time-stepping schemes on the simulation results is demonstrated at different inlet velocities. Tables 4.3 and 4.4 illustrate both the experimentally measured power of the Lux VAWT (see appendix C) compared to the average power output from the CFD simulations (using equations (1.3)–(1.6)) on the mesh M1 and M2 and their relative errors. The predicted average power of the Lux VAWT system appeared to lean towards the experimentally measured data when the density of the mesh improved, except for the fact that it creates a noticeable dynamic stall on the blades that eventually affects the power output (see Table 4.4). Also, the modified TR–BDF2 scheme produces an average power output that is somewhat comparable to the experimentally measured power and similar relative errors to other time-stepping schemes; this may be expected because all the methods are second-order accurate. Likewise, the average power output predicted by the CN scheme is less than the power output from the BDF2 scheme, which is independent of the time-step restriction and stiffness of the problem. In addition, the BDF2 scheme gives

the least relative error and predicts an average power that is closest to the measured data. This suggests the BDF2 method as the most suitable time-stepping scheme in simulating a transient flow for this application. Hence, the rest of the simulations were performed using the BDF2 scheme as a default time-stepping scheme.

Inlet	Measured	CN		BDF2		mTR–BDF2	
Vel. ($\mathrm{m\,s^{-1}}$)	Power	Power	Error (%)	Power	Error (%)	Power	Error (%)
4	2.6	6.17	137.00	6.19	138.3		
5	7.0	10.19	45.57	10.19	45.60		
6	13.5	13.45	0.37	13.48	0.15	13.68	1.33
7	21.8	22.25	2.09	22.34	2.46		
8	31.9	26.39	17.27	28.57	10.45	27.06	15.17
9	40.8	35.08	14.02	35.28	13.54		
10	47.2	40.21	14.81	40.43	14.34	41.89	11.25
11	49.2	44.01	10.55	44.27	10.20		
12	47.6	23.55	50.53	23.77	50.06	21.41	55.02
13	45.1	21.33	52.70	21.49	52.35		

Table 4.3: Average turbine power (kW) and relative error with various time-stepping schemes on mesh M1 at time-step $\Delta t = 0.00277$ s.

Inlet	Measured	BDF2	
Vel. ($\mathrm{m\,s^{-1}}$)	Power	Power	Error (%)
4	2.6	6.15	136.54
5	7	9.32	33.14
6	13.5	14.27	5.7
7	21.8	16.37	24.91
8	31.9	23.48	26.39
9	40.8	25.82	36.71
10	47.2	30.08	36.26
11	49.2	34.51	29.85
12	47.6	39.51	17.00
13	45.1	38.15	15.40
14	42.9	42.77	0.30
15	41.4	46.26	11.80
16	40.6	51.58	27.04
17	40.3	54.90	36.22
19	40.2	49.85	24.00
20	40.8	52.46	28.60

Table 4.4: Average turbine power (kW) and relative error with the BDF2 scheme on mesh M2 at time-step $\Delta t = 0.00277$ s.

Similarly, comparisons between the estimated power coefficient of the Lux VAWT from the CFD simulations using equation (1.7) and the experimentally measured data are shown in Table 4.5.

Inlet	TSR	Power Coefficient	
Vel. $(\mathrm{m\,s^{-1}})$	λ	**Exp.**	**CFD**
4	7.19	0.18	0.44
5	6.52	0.25	0.37
6	6.07	0.28	0.28
7	5.20	0.29	0.30
8	4.79	0.28	0.25
9	4.26	0.25	0.22
10	3.83	0.21	0.18
11	3.48	0.17	0.15
12	3.19	0.12	0.10
13	2.95	0.09	0.08
14	2.74	0.07	0.07
15	2.56	0.06	0.06
16	2.40	0.04	0.06
17	2.25	0.04	0.05
19	2.02	0.03	0.03
20	1.92	0.02	0.03

Table 4.5: Evaluation of power coefficient for the experimental and CFD simulation results of the Lux VAWT.

The table demonstrates the performance characteristics of the Lux VAWT under varying inflow and rotational conditions. Although the average efficiency for the turbine is somewhat above 25%, the efficiency generally varies relative to the wind speed or the tip speed ratio.

The performance of the wind turbines can generally be distinguished by a parameter known as the power coefficient. The power coefficient characterizes the capacity of a turbine to convert the kinetic energy of

the wind to electricity. This quantity expresses the ratio of the actual electric power generated by a wind turbine divided by the aggregate wind power flowing into the turbine at given wind speed. According to the Betz limit [14], the attainable peak value of the coefficient of performance is approximately 0.593. In practice, however, real coefficients of performance are less than the Betz limit due to various aerodynamic and mechanical losses [2]. The power coefficient of a turbine is relatively influenced by the unsteady aerodynamic forces, which depend on the pressure field around the turbine blades as well as the velocity profile of the blades. However, the turbine performance coefficient can also be affected by the blade design. During the turbine operation cycle, the aerodynamic forces exerted on the turbine blades vary and depend on the wind flow conditions, especially the relative velocity of the blade to the wind or the tip speed ratio. For a given turbine design, the coefficient of performance is measured relative to the tip speed ratio to quantify the effect of wind velocity on the power output of the turbine. With TSR variation, the rotor efficiency changes because the blade design cannot be optimal for every tip speed ratio.

Figure 4.2 illustrates the average power output from the CFD simulations and the experimentally measured data. The figure indicates the tip speed ratio or inlet velocity at which the power curve attains a maximum.

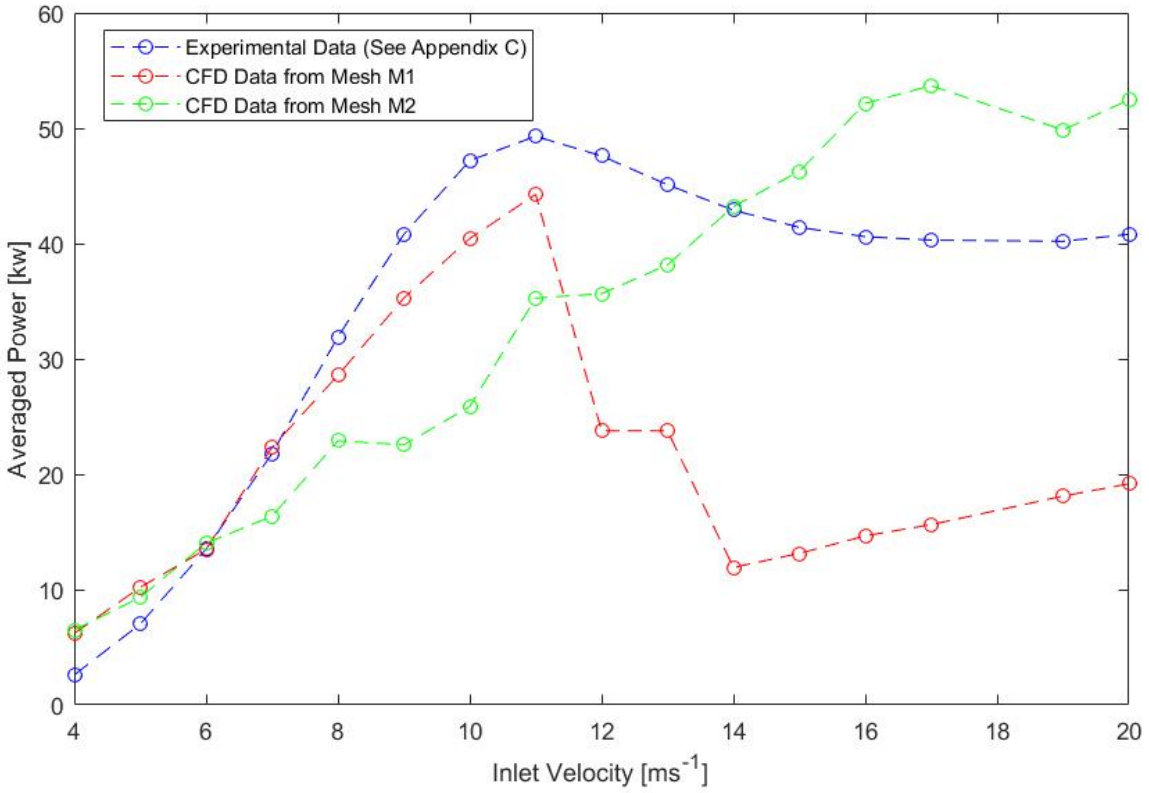
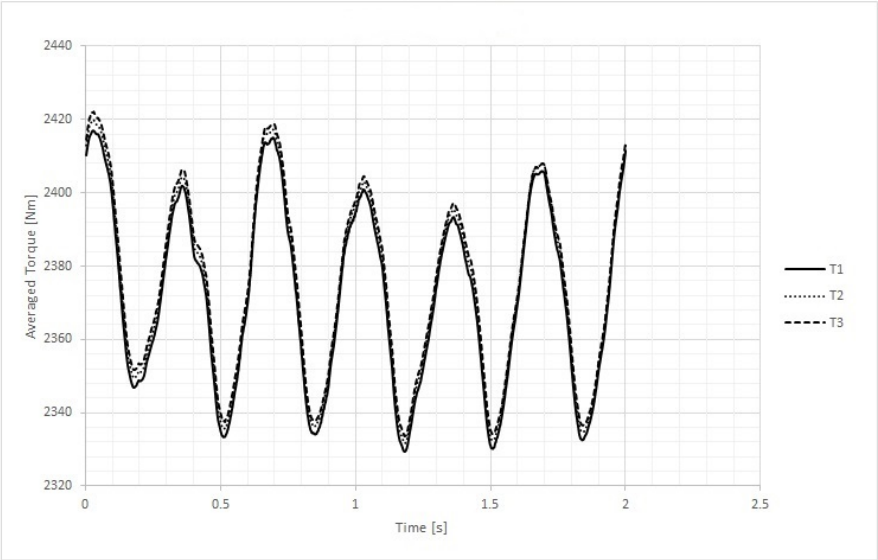


Figure 4.2: Comparison of the predicted power curve of the CFD solutions from mesh M1 and M2, and the experimental data.

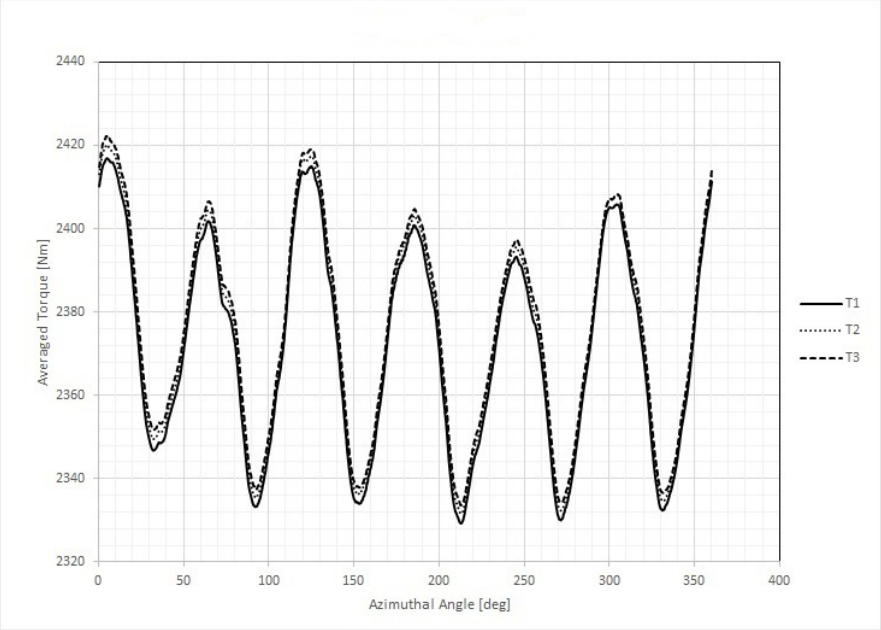
This power curve is obtained by plotting the average power output from the CFD simulations on mesh M1 and M2 against the tip speed ratio or the inlet velocities. The average output power is obtained as a product of the turbine instantaneous torque output (using equation (1.4)) and the angular speed of the rotor (using equation (1.2)), thus it is expected that the output power varies at different angular speeds. The power curve is generally valuable in estimating a turbine's ideal TSR, including its viability and adaptability to various aerodynamic conditions. With an increase in the wind speed above the cut-in speed of the Lux VAWT, there is a steady increase in the average power output. The power curve generated via the CFD simulations is quantitatively influenced by the excessive numerical diffusion and dispersion and did not coincide well with the corresponding experimental data. However, somewhere between 10 m/s – 12 m/s, the output power attains the limit of which the Lux VAWT is capable. This limit is often recognized as the rated power output, and the corresponding free-stream wind speed at which this occurs is known as the rated output wind speed. At higher wind speeds, however, the design of the turbine is arranged to constrain the output power to this maximum level such that there is no further increase to the output power.

4.2 Turbine Net Torque

The measurement of true mechanical torque generated by a wind turbine is a key parameter for the wind industry in design, control, and validation of the turbine systems. Torque evaluation holds promise as a valuable drivetrain condition-based monitoring variable to improve the turbine aerodynamic performance. Consequently, the resultant aerodynamic torque acting on the blades of the Lux VAWT is measured from the numerical computations with the standard OpenFOAM library. This net torque is evaluated at each time step of the computation with an in-built OpenFOAM library for force calculations (see Appendix D).
During each time step of the simulation, the net torque outputs change due to the rotation and the inlet velocity of the turbine. Figures 4.3–4.4 show the instantaneous torque output of the Lux VAWT at the quasi-steady solution with respect to the time-step and azimuthal angles of the flow. The aerodynamic torque is displayed when the turbine rotor was almost at uniform movement and the quasi-steady state solution is periodic. The curves reflect the presence of dynamic stall owing to a sudden drop in the generated lift (and by extension net torque) due to the sudden change in the angle of attack and the adverse pressure gradient of the flow. Specifically, around the azimuthal angle of 210° dynamic stall is observed due to a rapid drop in the net torque values with a noticeable increase in drag force.

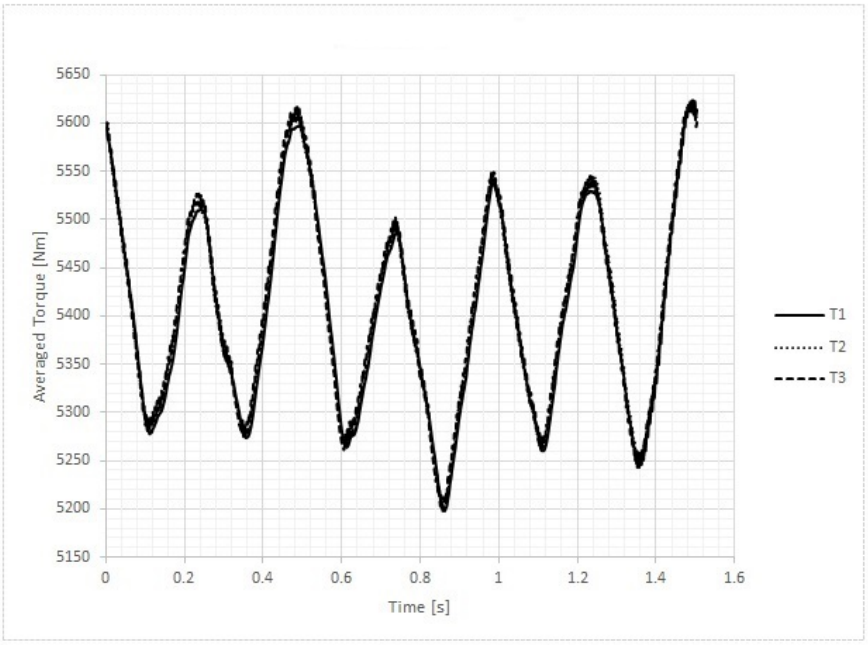


(a)

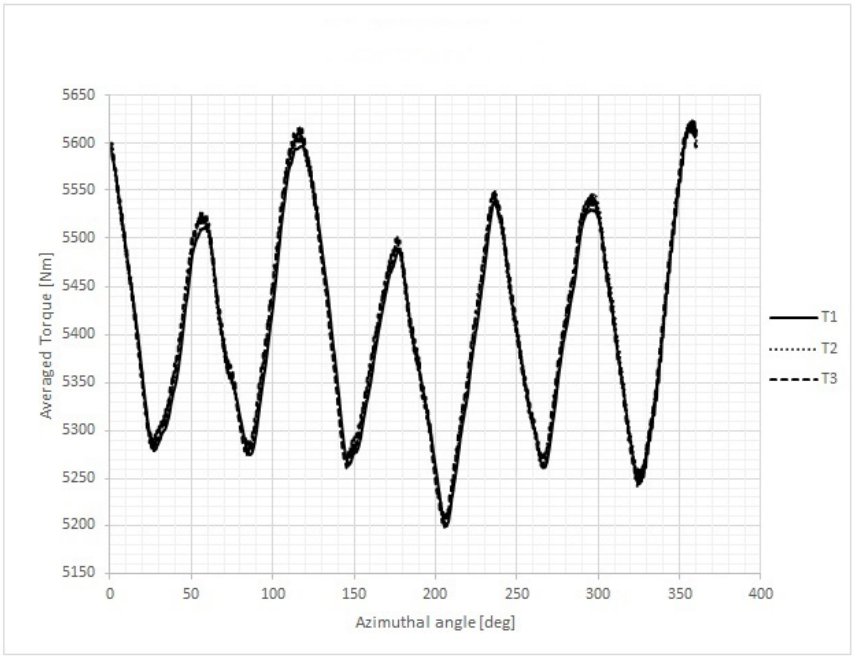


(b)

Figure 4.3: (a) Transient torque values at the inlet velocity of 4 m/s with respect to the simulation time for the last three revolutions of the simulation. (b) As for (a) but with regard to the azimuthal angle.



(a)



(b)

Figure 4.4: (a) Torque characteristic of the Lux VAWT at the TSR of 4.79 and inlet velocity of 8 m/s with respect to the simulation time for the last three revolutions of the simulation. (b) As for (a) but with regard to the azimuthal angle.

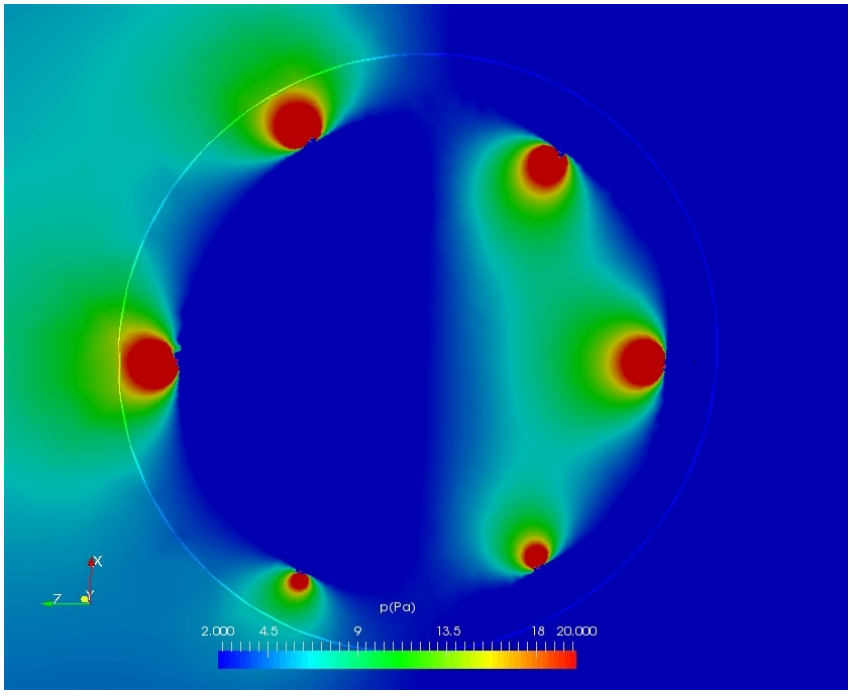
Furthermore, with six identical blades on the Lux VAWT, the frequency of the net torque is six times that of a single blade, and the plots are for the full cycle corresponding to the 360° of the azimuthal angle. From the graphs, it can be observed that the curves vary substantially with respect to the inlet velocity, an indication that the instantaneous torque distribution and force loading of each Lux VAWT blade changes all through the rotation cycle, and depends on the wind velocity in addition to other aerodynamic factors. The differences in the net torque and lift force generation by the blades were nevertheless small throughout the rotation cycle. For example, the net torque distribution is low at the cut-in speed of 4 m/s (see Figure 4.3), which corresponds to the highest TSR. The averaged net torque range from 2340 Nm to about 2420 Nm. This range becomes more narrow as the inlet velocity increases. As the TSR decreases (with increased free-stream velocity), the maximum averaged net torque increases, and the location of the peak net torque moves to 120° from 100°. Similar trends were observed in Figure 4.4, with the instantaneous torque output increases gradually going towards the lowest TSR as incoming wind velocity of the Lux VAWT increases. In addition, the variation of torque output during the accelerating and decelerating flows further emphasize the significance of transient phenomena on the aerodynamic performance of the Lux VAWT.

4.3 Pressure Contour

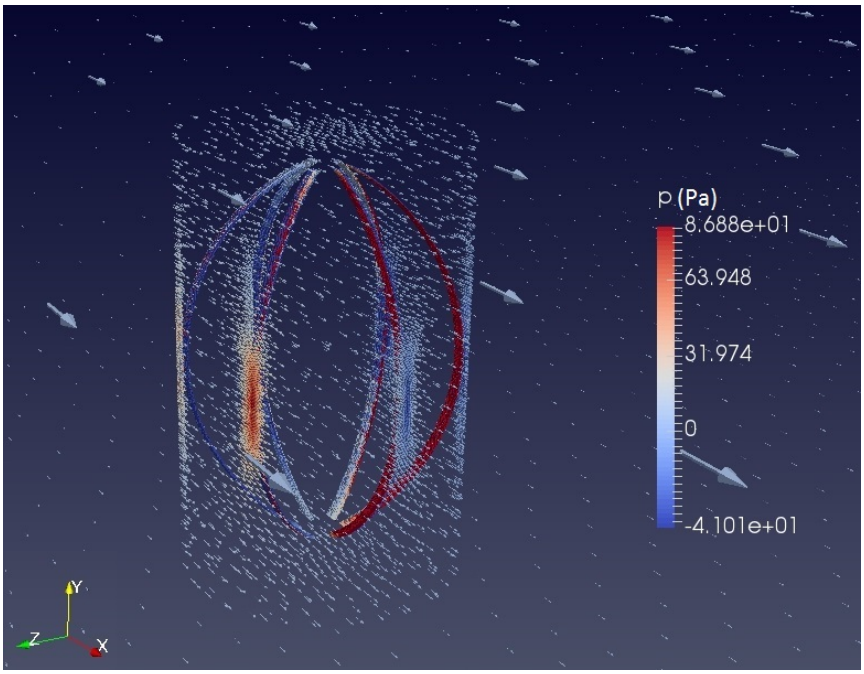
Having shown the graphical illustrations of the Lux VAWT net torque relative to the wind free-stream speed, the flow visualization is examined to provide information about the structure of the wakes and flow separation. Flow visualization facilitates the process of identifying the performance limiting factors at each instance of the simulations with respect to the tip speed ratio. In this respect, Figures 4.5 – 4.6 demonstrate the contours of highly non-uniform pressure distributions on the blade surfaces, rotating domain, and stationary domain of the Lux VAWT at two different inlet velocities. At the instance of blade rotation, there is an observable pressure difference between the pressure side and the suction side of the blades. As illustrated in Figure 4.5a, a greater differential of the pressure distributions is observed on the blade surfaces, where the turbulent kinetic energy transfer occurred, leading to more power generation. High pressure regions are noticed within the rotating domain and on the surfaces of the blades, whereas moderate pressure profiles are observed at the rear end with significant reduction within the wind turbine wake. Due to the prevalence of this differential pressure, the blade thrust forces are in essence greater than the drag, thus causing the turbine rotor to rotate continuously and make maximum contribution to the instantaneous torque generation. The observed pressure differential is due primarily to the three-dimensional rotatory effect, dynamic flow separation on the blades, and the angle of attack of the blades' surfaces to the incoming free-stream wind velocity [21].
In Figures 4.5b - 4.5c particularly, there is a noticeable variation in the pressure distributions on the blades. It is noticed that there is a gradual increase in the pressure distributions along the blade surfaces from the upstream side of the rotating domain to the downstream side. Also, we observed that the negative side of the camber had lower pressure distributions compared to the positive camber. The positive and negative

pressure distribution regions and their difference accounts for the turbine rotation. Figure 4.5b indicates that the flow separation has not fully occurred on the Lux VAWT blades leaving large portions at lower pressures than are present at the 12 s of the simulation time in Figure 4.5c, where there is a high pressure region at the mid-span section of the blades. The rationale for this observation is that the enclosed surface of the blade's lateral part reduces the cross-sectional area of the flow field within the rotating domain, incites an increased airflow rate, and eventually, weakens the observable pressure field.

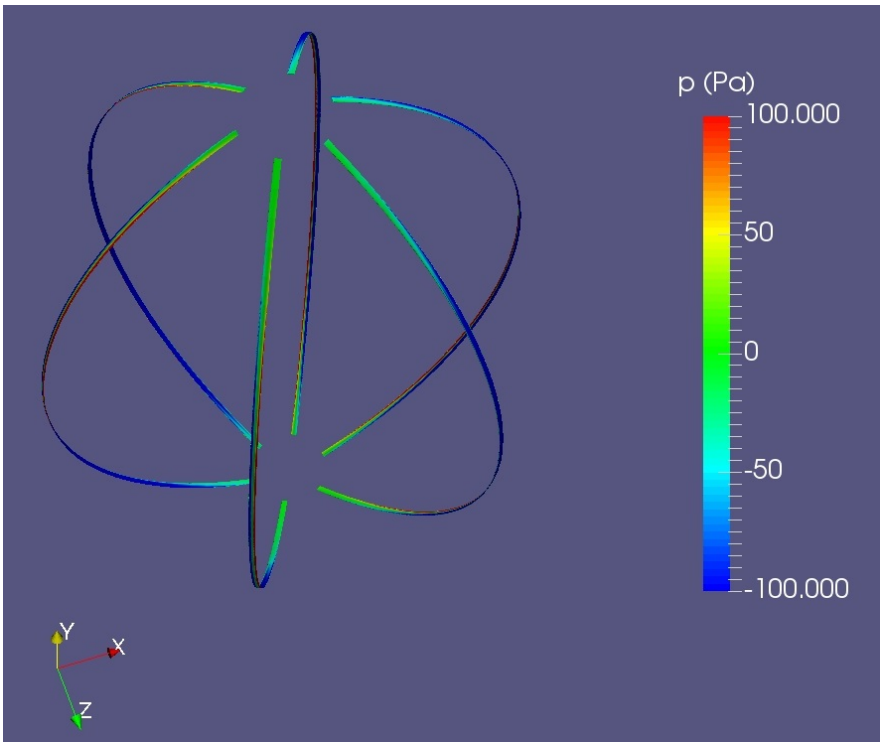
Similarly, Figure 4.6 suggests a comparable pattern of pressure contours distribution within the computational domain and the rotating section of the Lux VAWT at 7 m/s free-stream velocity. The figure characterizes the build-up of kinetic energy on the blade as a function of pressure distribution within the rotating domain. The contour profile as captured in Figure 4.6a shows substantial changes in the pressure as the wind flows near the inlet of the stationary domain towards the blade regions and at the exit of the rotating domain with low velocity. The pressure soon after decreases as the wind exits the rotating domain towards the outlet region, apart from where it is presently funneled at the blades. The wind energy coming through the inlet region is partially obstructed by the turbine blades, which acquire some of the kinetic energy. By comparison, the reduced wind speed creates minimal pressure value on the blade surfaces and within the flow domain than the higher wind speed. The pressure drops steadily due to the fluid flow velocity that gradually decreases after colliding with the surface of the blades. In general, the front side of the blades has a higher pressure distribution than the corresponding rear side. This static pressure distribution is responsible for the turbine net torque generation.



(a)

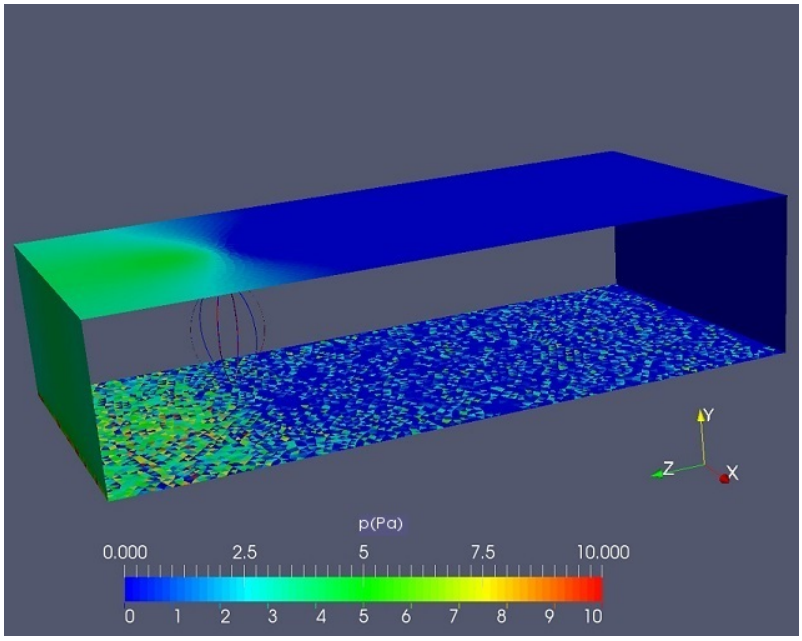


(b)

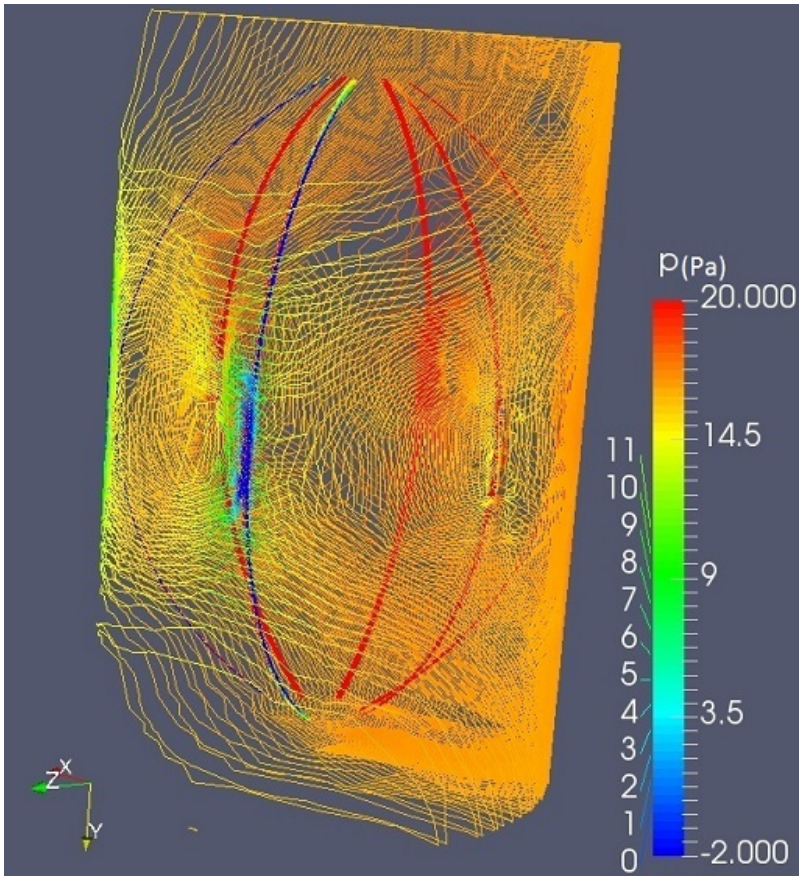


(c)

Figure 4.5: (a) The pressure contour around the Lux VAWT blades section (NACA0012 airfoil). (b) The pressure distribution on the surface of the Lux turbine blades within the rotating domain for an inlet velocity of 9 m/s at $t = 6$ s (c) As for (b) but when the CFD solution reaches a quasi-steady state.



(a)



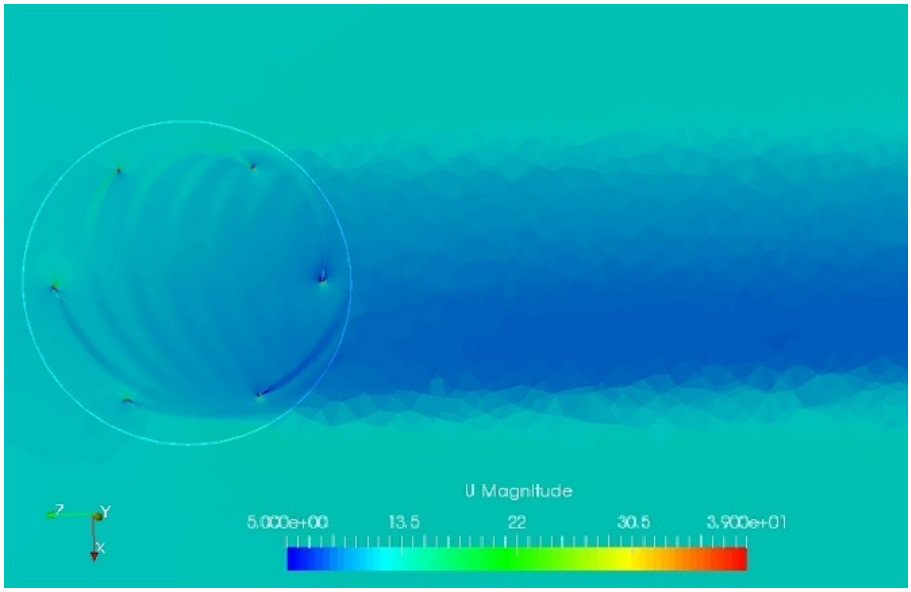
(b)

Figure 4.6: (a) The pressure contour distribution on the whole domain for an inlet velocity of 7 m/s at $t = 6$ s. (b) The pressure distribution at the quasi-steady of the CFD solution within the rotating domain.

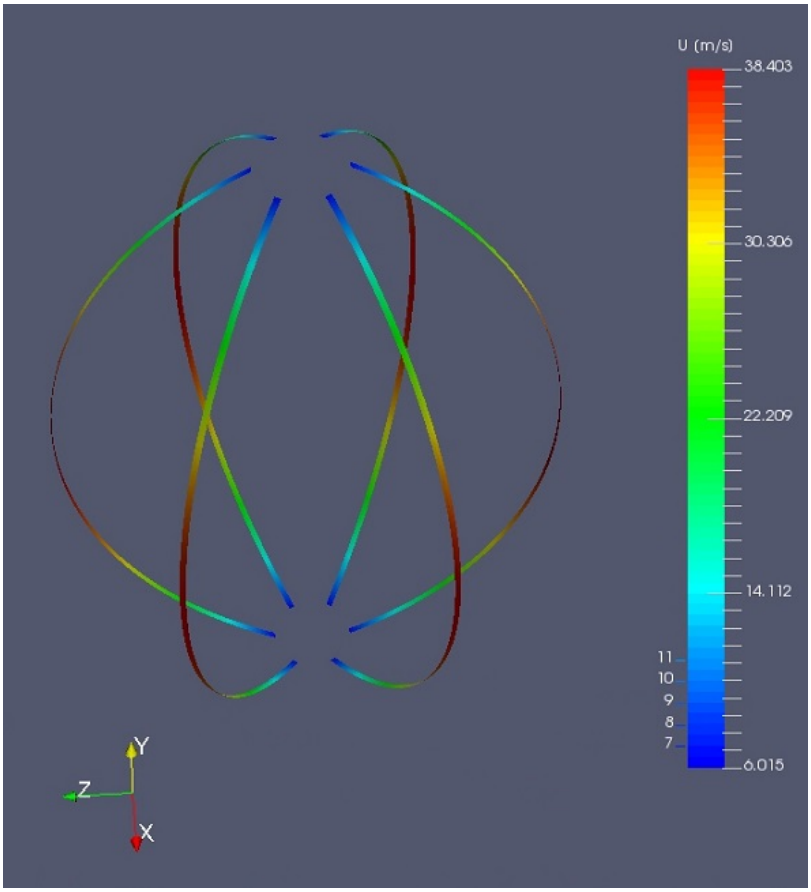
4.4 Velocity Contour

Contours of the velocity magnitude on the turbine blades are displayed in Figure 4.7 - 4.8 for the inlet velocity 9 m/s and 7 m/s respectively. From Figure 4.7a, there seems to be higher velocity magnitude around the turbine blades than the remainder of the computational domain. In addition, the velocity magnitudes around the turbine blades, especially in the upwind side of the turbine, are substantially higher compared to the leeward side of the turbine. Furthermore, the flow profile within the center of the rotating domain and the rest of the stationary region reflects nonuniform velocity distributions, with a significant loss of velocity in the wake of the turbine. This can be attributed to the outstanding difference between the free-stream wind velocity and the tangential velocity of the blades as they drift windward.

Also in Figure 4.7b, there is a steady increase in the velocity profile near the mid-span section of the blade surfaces due to high turbulent flow formation within the rotating region. The increment in the velocity distributions impacts the aerodynamic forces acting along the lateral section of the turbine blades and constitutes a greater pressure difference that boost force generation on the blades. This explains why the curve of instantaneous torque was higher at higher inlet velocities.



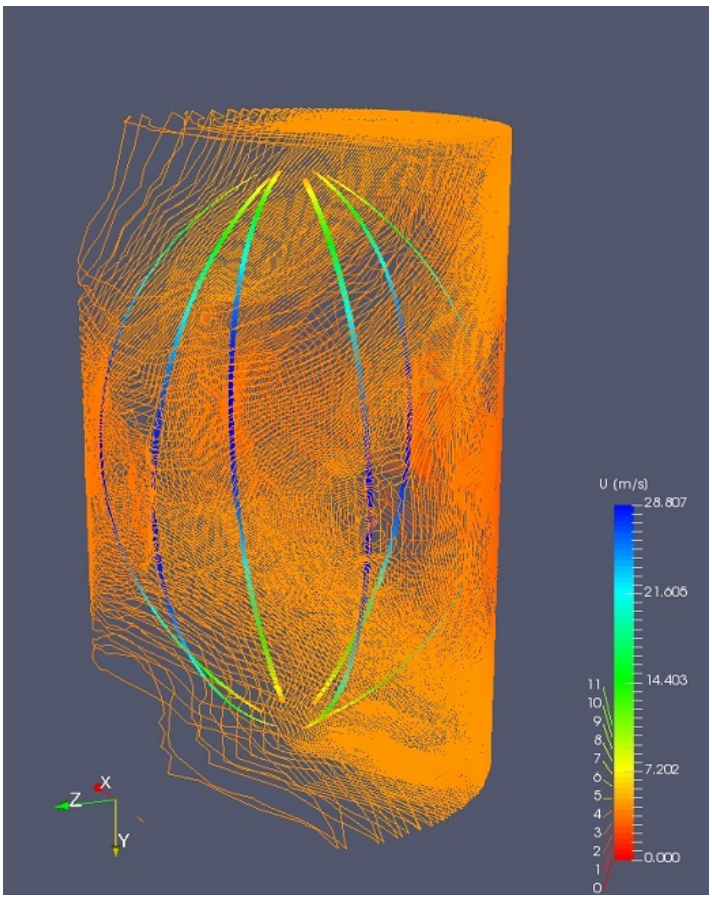
(a)



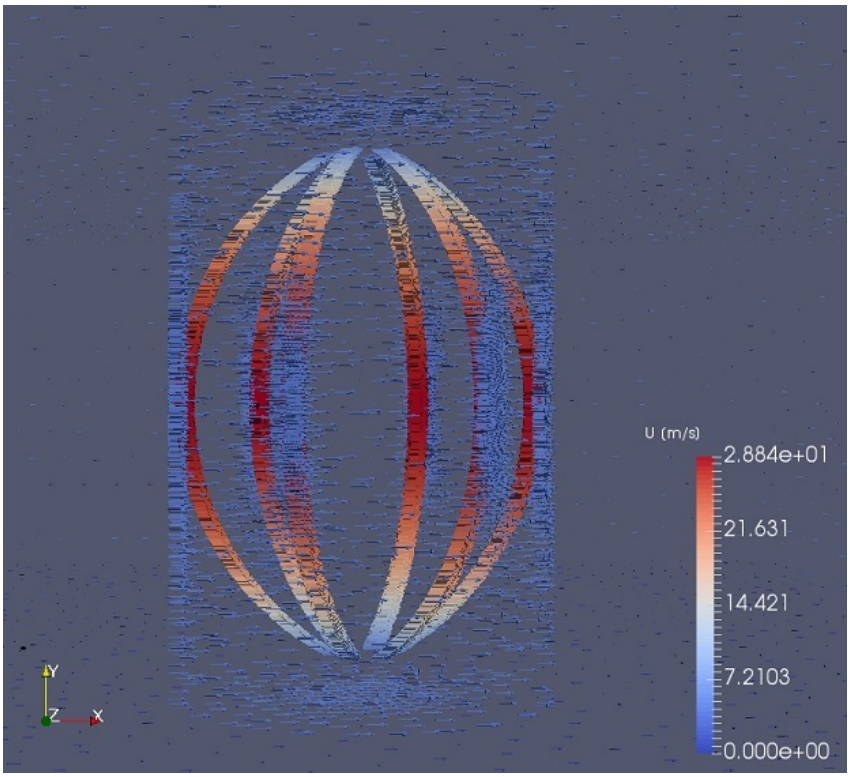
(b)

Figure 4.7: (a) The variation of the velocity magnitude in the computational domain. (b) The velocity contour on the Lux VAWT blades during the inlet velocity of 9 m/s at $t = 6$ s.

Similarly, in Figure 4.8a, the velocity distribution on the turbine blades is stronger than what is observed in the downstream area. Because of the rotatory effect, the velocities from the leading edge increased towards the mid-span of each blade. However, the velocity of wake away from the turbine blades is smaller than the velocity of the upstream air flow. The low-speed region, which is observed behind the hub of the rotating blades, characterizes the wake formation. This occurrence can be identified with the generation of a positive pressure gradient on the two sides of the turbine blades despite the contribution of the rotatory effect, which is explicitly evident from the extension of the higher velocity at the mid-span side of the turbine blades (see Figure 4.8b).



(a)



(b)

Figure 4.8: (a) The velocity magnitude contour in the rotating domain for the inlet velocity of 7 m/s at $t = 6$ s. (b) The velocity vectors and contours of velocity magnitude and blade's circulation for an inlet velocity of 7 m/s at $t = 12$ s.

5 Conclusion and Recommendations

5.1 Conclusion

The main purpose of this thesis is to develop a simulation model to investigate the Lux VAWT aerodynamic performance. This simulation model enables us to quantitatively study the behavior of the turbine and examines the practicality of the AMI dynamic (sliding) mesh technique to predict the aerodynamic flow around the Lux VAWT at different aerodynamic conditions. The aerodynamics parameters, which influence the performance characteristics of the Lux VAWT, have been studied to understand some of the geometrical features and operations of the turbine. Rather than depend on the wind tunnel experiments that can be costly, risky, and easily influenced by the instability and interference in evaluating the performance characteristics of the wind turbine, one can undertake CFD simulations to produce cost-effective wind turbine performance assessment. With the use of the k–ω SST turbulence model in OpenFOAM and a mesh that achieves an adequate y^+(cell wall distance) near the blade domain in a multiple reference frame environment, the turbine performance can be predicted in terms of power output while the flow conditions can be studied.

Despite the fact that not all the expected tasks have been fully accomplished, the results presented have been useful to validate the working of the moving mesh CFD codes (such as PimpleDyMFoam and modified TR-BDF2 scheme) and the measured data from the experiment on the Lux VAWT system. The generation of a mesh sufficient to get acceptable results has taken a significant part of our numerical study. Power estimations were evaluated and compared to the experimentally measured data. Some of the results did not match well with the experimentally measured data. In addition, when investigating the performance characteristics of a turbine system in structured conditions, the numerical simulations must be cautiously regulated in terms of turbulence models, wall treatment models, and discretization schemes. In the presented results, it is clear that for the complex flow simulation of the Lux VAWT, that temporal discretization schemes have impact on the results. After studying the simulations and post-processing all the generated and calculated data, the observations are:

- The results analyzed prior to 10 revolutions of the turbine lead to large overestimation of the net torque. Thus, the number of revolutions sufficient to realize a steady solution is 10-16, where the difference in net torque values between 10 and 16 revolutions is less than 2%.

- As the inlet velocity of the wind increases, the power output increases until a certain inlet velocity, at

which the power output begins to decline.

- The average power of the Lux VAWT and the estimated net torque values are directly correlated and share a similar trend. The higher the TSR (lower wind free-stream velocity), the better the power coefficient of the Lux VAWT.
- The maximum instantaneous torque value is attained around the azimuthal position 120°.
- A finite volume discretization approach with sufficiently low y^+ is important to capture all the solution variable gradients within the turbulent boundary layer and accurately predict the average power output of the turbine.

5.2 Possible future research directions

The results of this study can be improved in many ways, specifically in the areas of discretization accuracy, sensitivity analysis, and computational domain mesh refinement. Accordingly, future research directions are suggested in order to improve the preliminary findings. Most importantly, it seems to be necessary to:

- perform numerical simulations designed at
  - reducing numerical diffusion when triangular meshes are interchanged by either structured meshes or polyhedral meshes within the rotatory region of the turbine.
  - increasing the accuracy of CFD estimates when other two-equation nonlinear eddy viscosity models (turbulence models) and wall treatment models that are capable of capturing turbulence anisotropies without increasing the computational effort are applied.
- assess the quantitative effects of each blade interaction on the flow dynamics and the overall performance characteristic of the turbine. Similarly, knowing the exact torque value generated at every azimuthal angle with time can provide an in-depth understanding on the self-start capability of the turbine and the development of design control systems to optimize performance.
- optimize the characteristic performance of the Lux VAWT by varying the angle of attack, aspect ratio, airfoil shape (camber and thickness effects), rotor design, and twist angle for the optimal blade configuration and performance.
- perform sensitivity analysis on the Lux VAWT geometry to further study the effect of domain size and inlet distance on the predicted power performance and stream-wise velocity profile of the turbine.

References

[1] N. Abasi, M. Suleiman, N. Abbasi, and H. Musa. Two-point block BDF method with off-step points for solving stiff ODEs. *Journal of Soft Computing and Applications*, 2014:1–15, 2014.

[2] J. S. Ahad and A. Shakil. Hypothetical discussion on the windmill on train. *2013 International conference on electrical information and communication technology*, 1:1–9, 2014.

[3] J. O. Ajedegba. Effects of blade configuration on flow distribution and power output of a zephyr vertical axis wind turbine. Master's thesis, University of Ontario institute of technology, 2008.

[4] A. Ali, H. Chowdhury, B. Loganathan, and F. Alam. An aerodynamic study of a domestic scale horizontal axis wind turbine with varied tip configurations. *Procedia Engineering*, 105(2015):757–762, 2015.

[5] I. Ammara, C. Leclerc, and C. Masson. A viscous three-dimensional differential actuator-disk method for the aerodynamic analysis of wind farms. *Journal of Solar Energy Engineering*, 124(4):345–356, 2002.

[6] J. D. Anderson. *Computational Fluid Dynamics: The Basics with Applications.* McGraw-Hill Education; 1st edition, New York, USA, 1995.

[7] B. Andersson, R. Andersson, L. Hkansson, M. Mortensen, R. Sudiyo, and B. van Wachem. *Computational Fluid Dynamics for Engineers.* Cambrigde University Press; 1st edition, 2012.

[8] ANSYS. ANSYS CFX-Solver Theory Guide (2011) and ANSYS ICEM CFD User Manual (12.1). `http://www.ansys.com`, 2009. Online; accessed 15 May 2018.

[9] J. Antonio, V. Alé, M. Petry, P. Sérgio, B. Garcia, G. Cirilo, S. Simioni, and G. Konzen. Performance evaluation of the next generation of small vertical axis wind turbine. *European Wind Energy Conference and Exhibition, May 7-10, MIC - Milano Convention Centre, Milan, Italy*, 2007.

[10] R. E. Bank, W. M. Coughran, W. Fichtner, E. Grosse, D. J. Rose, and R. K. Smith. Transient simulation of silicon devices and circuits. *IEEE Transactions on Electron Devices*, 32:1992–2007, 1985.

[11] G. K. Batchelor. *An Introduction to Fluid Dynamics.* Cambridge University Press, Cambridge, 1967.

[12] L. Battisti, L. Zanne, S. Dell-Anna, V. Dossena, G. Persico, and B. Paradiso. Aerodynamic measurements on a vertical axis wind turbine in a large scale wind tunnel. *Journal of Energy Resources Technology*, 133(3):31201–31209, 2011.

[13] T. Behrens. Openfoam's basic solvers for linear systems of equations. *Chalmers, Department of Applied Mechanics*, 18(2), 2009.

[14] A. Betz. *Introduction to the Theory of Flow Machines. (D. G. Randall, Trans.).* Oxford: Pergamon Press, 1966.

[15] M. M. A. Bhutta, N. Hayat, A. U. Farooq, Z. Ali, S. R. Jamil, and Z. Hussain. Vertical axis wind turbine–A review of various configurations and design techniques. *Renewable and Sustainable Energy Reviews*, 16(4):1926–1939, 2012.

[16] A. Biswas, R. Gupta, and K. K. Sharma. Experimental investigation of overlap and blockage effects on three-bucket Savonius rotors. *Wind Engineering*, 31(5):363–368, 2007.

[17] L. Bonaventura and A. D. Rocca. Monotonicity, positivity and strong stability of the TR-BDF2 method and of its SSP extensions. *arXiv e-prints*, 56, 2015.

[18] J. Boussinesq. Essay on the theory of running waters. *Memoirs presented by various scholars at the Academy of Sciences*, 23:1–680, 1877.

[19] T. Brahimi, A. Allet, and I. Paraschivoiu. Aerodynamic analysis models for vertical-axis wind turbines. *International Journal of Rotating Machinery*, 2(1):15–21, 1995.

[20] A. Busse, M. Thakkar, and N. D. Sandham. Reynolds number dependence of the near-wall flow over irregular rough surfaces. `https://pdfs.semanticscholar.org/0f63/de3f559a41a497a5c2d721fc169740c50848.pdf`. Online; accessed 10 March 2019.

[21] C. P. Butterfield, D. Simms, G. Scott, and A. C. Hansen. Dynamic stall on wind turbine blades. *No. NREL/TP-257-4510; CONF-9109112–7. National Renewable Energy Lab. Colorado.*, 1991.

[22] M. R. Castelli, G. Ardizzon, L. Battisti, E. Benini, and G. Pavesi. Modeling strategy and numerical validation for a Darrieus vertical axis micro-wind turbine. *ASME 2010 International Mechanical Engineering Congress and Exposition*, 7:409–418, 2010.

[23] M. R. Castelli and E. Benini. Comparison between lift and drag-driven vawt concepts on low-wind site AEO. *International Journal of Environmental and Ecological Engineering*, 5(11):669–673, 2011.

[24] A. M. Chowdhury, H. Akimoto, and Y. Hara. Comparative CFD analysis of vertical axis wind turbine in upright and tilted configuration. *Renewable energy*, 85:327–337, 2016.

[25] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100(1):32–74, 1928.

[26] R. Courant, E. Isaacson, and M. Rees. On the solution of nonlinear hyperbolic differential equation by finite diferences. *Comm. Pure and Appl. Math.*, 5:243–255, 1952.

[27] J. Crank and P. Nicolson. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Mathematical Proceedings of the Cambridge Philosophical Society*, 43(1):50–67, 1947.

[28] R. J. Crossley and P. J. Schubel. Wind turbine blade design. *Energies*, 5(9):3425–3449, 03 2012.

[29] D. Dutykh. How to overcome the Courant-Friedrichs-Lewy condition of explicit discretizations? `https://hal.archives-ouvertes.fr/hal-01401125/document`.

[30] D. N. Arnold. Stability, consistency, and convergence of numerical discretizations. `http://www-users.math.umn.edu/~arnold/papers/stability.pdf`, 2015. Online; accessed 15 April 2018.

[31] A. Das and P. K. Talapatra. Modelling and analysis of a mini vertical axis wind turbine. *International Journal of Emerging Technology and Advanced Engineering*, 6(6):184–194, 2016.

[32] L. Davidson. *An introduction to turbulence models.* Chalmers university of Technology, Gothenburg, Sweden, 2003.

[33] M. Douak and Z. Aouachria. Starting Torque Study of Darrieus Wind Turbine. *International Journal of Physical and Mathematical Sciences*, 9(8):476–481, 2015.

[34] E. A. Cowen. Canonical Turbulent Flows. `http://www.ceeserver.cee.cornell.edu/eac20/cee637/handouts/TURBFLOW_L8.pdf`.

[35] E. G. Kadlec. Characteristics of Future Vertical-Axis Wind Turbines (SAND78-1068). `http://prod.sandia.gov/techlib/access-control.cgi/1979/791068.pdf`, 1978. Online; accessed 05 March 2018.

[36] S. Eriksson, H. Bernhoff, and M. Leijon. Evaluation of different turbine concepts for wind power. *Renewable and Sustainable Energy Reviews*, 12(5):1419–1434, 2008.

[37] F. P. Karrholm. Rhie-Chow interpolation in OpenFOAM. `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2007/rhiechow.pdf`, 2006.

[38] C. S. Ferreira, H. Bijl, G. V. Bussel, and G. V. Kuik. Simulating dynamic stall in a 2D VAWT: Modeling strategy, verification, and validation with particle image velocimetry data. *Journal of Physics: Conference Series*, 75(1):12–23, 2007.

[39] J. H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer-Verlag Berlin Heidelberg; 3rd ed. New York, 2002.

[40] C. Fredebeul. A BDF: A generalization of the backward differentiation formulae. *SIAM Journal on Numerical Analysis*, 35(5):1917–1938, 1998.

[41] G. Ceetron. GLview Inova Software Suite: The GLview Visualization Concept. `https://www.smartcae.com/software/ceetron/glview-inova/`.

[42] G. Lux. Lux Wind Turbines System. `http://www.luxwindturbines.com/technology.html`, 2009.

[43] B. Galperin. *Large Eddy Simulation of Complex Engineering and Geophysical Flows*. Cambridge University Press; 1st edition, 2010.

[44] GE Energy Consulting. Pan-Canadian Wind Integrated Study (PCWIS): Report for canadian wind energy association (CanWEA). `https://canwea.ca/wind-integration-study/full-report/`. Online; accessed 24 April 2017.

[45] C. W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice Hall; 1st edition, Englewood-Cliffs, NJ, 1971.

[46] R. Gospinath and V. Ganesan. Orthogonal arrays: An introduction and their application in optimizing underrelaxation factors in SIMPLE-based algorithm. *International Journal for Numerical Methods in Fluids*, 14(6):665–680, 1992.

[47] R. Gross, M. Leach, and A. Bauen. Progress in renewable energy. *Environment International*, 29(1):105–122, 2003.

[48] B. Guo, T. A. Langrish, and D. F. Fletcher. An assessment of turbulence models applied to the simulation of a two-dimensional submerged jet. *Applied Mathematical Modelling*, 25(8):635–653, 2001.

[49] R. Gupta, A. Biswas, and K. K. Sharma. Comparative study of a three-bucket Savonius rotor with a combined three-bucket Savonius–three-bladed Darrieus rotor. *Renewable Energy*, 33(9):1974–1981, 2008.

[50] GWEC. Global Wind Statistics Report by Global Wind Energy Council. `https://www.gwec.net/wp-content/uploads/2013/02/GWEC-PRstats-2012-english.pdf`, 2012. Online; accessed 10 March 2017.

[51] H. Kudela. Turbulent flow. `http://www.itcmp.pwr.wroc.pl/~znmp/dydaktyka/fundam_FM/Lecture_no3_Turbulent_flow_Modelling.pdf`.

[52] S. Ha, J. Park, and D. You. A GPU-accelerated semi-implicit fractional-step method for numerical solutions of incompressible Navier–Stokes equations. *Journal of Computational Physics*, 352:246–264, 2018.

[53] B. Habtamu and Y. Yingxue. Effect of camber airfoil on self starting of vertical axis wind turbine. *Journal of Environmental Science and Technology*, 4(3):302–312, 2011.

[54] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, page 614. Springer-Verlag, Berlin Heidelberg, 2nd edition, 1996.

[55] P. R. Hammer. *Computational study on the effect of Reynolds number and motion trajectory asymmetry on the aerodynamics of a pitching airfoil at low Reynolds number*. PhD thesis, Michigan State University, 2016.

[56] E. Hau. *Wind Turbines: Fundamentals, Technologies, Application, Economics*. Springer-Verlag Berlin Heidelberg; 2nd ed., 2006.

[57] Herbert J. Sutherland and Dale E. Berg and Thomas D. Ashwill . A Retrospective of VAWT Technology (SAND2012-0304). `http://prod.sandia.gov/techlib/access-control.cgi/1979/791068.pdf`, 2012. Online; accessed 05 May 2018.

[58] J. O. Hinze. *Turbulence*. McGraw-Hill College, 2nd edition, 1975.

[59] C. Hirsch. *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics*. Butterworth-Heinemann; 2nd edition, 2007.

[60] M. E. Hosea and L. F. Shampine. Analysis and implementation of TR-BDF2. *Applied Numerical Mathematics*, 20(1–2):21–37, 1996.

[61] R. Howell, N. Qin, J. Edwards, and N. Durrani. Wind tunnel and numerical study of a small vertical axis wind turbine. *Renewable Energy*, 35(2):412–422, 2010.

[62] A. Iserles. *A first course in the numerical analysis of differential equations*. Cambrigde University Press; 1st edition, 1996.

[63] M. Islam, D. S. K. Ting, and A. Fartaj. Aerodynamic models for Darrieus-type straight-bladed vertical axis wind turbines. *Renewable and Sustainable Energy Reviews*, 12(4):1087–1109, 2008.

[64] R. I. Issa. Solution of the implicitly discretized fluid flow equations by operator-splitting. *J. Comp. Physics*, 62:40–65, 1986.

[65] J. Kortelainen. Meshing Tools for Open Source CFD – A Practical Point of View. `http://https://www.vtt.fi/inf/julkaisut/muut/2009/VTT-R-02440-09.pdf`.

[66] J. Zayas. Wind Vision: United States department of energy wind vision report. `https://energy.gov/eere/wind/wind-vision`, 2016. Online; accessed 21 November 2017.

[67] A. James, G. Berk, and L. Charles. Paraview: An end-user tool for large data visualization. *The Visualization Handbook-Energy*, 836:713–717, 2005.

[68] H. Jasak. *Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows*. PhD thesis, University of London and Diploma of Imperial College, 1996.

[69] A. R. Jha. *Wind Turbine Technology*. CRC Press; 1st edition, 2010.

[70] W. Jones and B. Launder. The prediction of laminarization with a two-equation model of turbulence. *International Journal of Heat and Mass Transfer*, 15(2):301 – 314, 1972.

[71] K. G. Pierce. Flow in pipes. `https://www.uio.no/studier/emner/matnat/math/MEK4450/h11/undervisningsmateriale/modul-5/Pipeflow_intro.pdf`, 2004. Online; accessed 27 August 2018.

[72] K. Stüben. Algebraic Multigrid AMG: An Introduction with Applications. `https://www.scai.fraunhofer.de/content/dam/scai/de/documents/AllgemeineDokumentensammlung/SchnelleLoeser/SAMG/AMG_Introduction.pdf`, 1999. Online; accessed 5 January 2019.

[73] G. Kalitzin, G. Medic, G. Iaccarino, and P. Durbin. Near-wall behavior of RANS turbulence models and implications for wall functions. *Journal of Computational Physics*, 204:265–291, 2005.

[74] A. Katz and V. Sankaran. Mesh quality effects on the accuracy of CFD solutions on unstructured meshes. *Journal of Computational Physics*, 230(20):7670–7686, 2011.

[75] R. Koomullil, B. Soni, and R. Singh. A comprehensive generalized mesh system for CFD applications. *Mathematics and Computers in Simulation*, 78(5–6):605–617, 2008.

[76] S. Kotsiantis and D. Kanellopoulos. Discretization techniques: A recent survey. *GESTS International Transactions on Computer Science and Engineering*, 32(1):47–58, 2006.

[77] P. A. Kozak, D. Vallverdu, and D. Rempfer. Modeling vertical-axis wind turbine performance: Blade element method vs finite volume approach. *Journal of Propulsion and Power*, 32(3):592–601, 2016.

[78] R. Kumar and P. Baredar. *Solidity study and its effects on the performance of a small scale horizontal-axis wind turbine*. Excellent Publishing House, 2012.

[79] L. R. Hellevik. Numerical Methods for Engineers, Department of Structural Engineering, NTNU. `http://folk.ntnu.no/leifh/teaching/tkt4140/._main061.html`.

[80] J. Lambert. *Computational Methods in Ordinary Differential Equations*. Wiley Press; 1st edition London, 1973.

[81] L. Landau and E. Lifchitz. *Fluid Mechanics*. Pergamon Press, Paris, 1959.

[82] R. Lapuh. *Mesh Morphing Technique used with Open-Source CFD Toolbox in Multidisciplinary Design Optimisation*. PhD thesis, Department of Information Technology, Uppsala Universitet, 2018.

[83] B. E. Launder. First steps in modelling turbulence and its origins: a commentary on reynolds (1895) on the dynamical theory of incompressible viscous fluids and the determination of the criterion. *Philos Trans A Math Phys Eng Sci*, 373(2039):1–12, 2015.

[84] B. E. Launder and D. B. Spalding. The numerical computation of turbulent flows. *Computer Methods in Applied Mechanics and Engineering*, 3:269–289, 1974.

[85] M. Lesieur. *Turbulence in Fluids: Stochastic and Numerical Modeling*. Springer 2nd edition, Netherlands, Dordrecht, 1990.

[86] R. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics (SIAM), 1st edition, 2007.

[87] C. C. Lin. *On the instability of laminar flow and its transition to turbulence : In Grtler H. (eds) Boundary Layer Research. International Union of Theoretical and Applied Mechanics. Springer, Berlin Heidelberg*. Springer, Berlin Heidelberg, 1958.

[88] M. R. Emami. Aerodynamics Forces on an Airfoil. `http://www.aerospace.utoronto.ca/pdf_files/open_subsonic.pdf`. Online; accessed 7 March 2019.

[89] M. Ragheb. Vertical Axis Wind Turbine. `http://www.mragheb.com/NPRE%20475%20Wind%20Power%20Systems/Vertical%20Axis%20Wind%20Turbines.pdf`, 2015.

[90] S. MacLachlan, J. Moulton, and T. Chartier. Robust and adaptive multigrid methods: comparing structured and algebraic approaches. *Numer. Linear Algebra Appl.*, 19(2):389–413, 2012.

[91] J. F. Manwell, J. G. McGowan, and A. L. Rogers. *Wind Energy Explained: Theory, Design, and Application*. John Wiley and Sons; 2nd edition, 2010.

[92] P. McKay, R. Carriveau, and D. S. K. Ting. Wake impacts on downstream wind turbine performance and yaw alignment. *Wind Energy*, 16(2):221–234, 2013.

[93] F. R. Menter. Two-equation eddy-viscosity turbulence models for engineering aplications. *AIAA J*, 32:1598–1605, 1994.

[94] F. R. Menter, M. Kuntz, and R. B. Langtry. Ten years of industrial experience with the SST turbulence model. *Turbulence, Heat and Mass Transfer*, 4:625–632, 2003.

[95] Michael Taylor. International Renewable Energy Agency (IRENA): Report on Renewable Power Generation Costs in 2017. `https://www.irena.org/-/media/Files/IRENA/Agency/Publication/2018/Jan/IRENA_2017_Power_Costs_2018_summary.pdf?la=en&hash=6A74B8D3F7931DEF00AB88BD3B339CAE180D11C3`, 2018. Online; accessed 10 August 2018.

[96] R. Mosfequr, K. N. Morshed, J. Lewis, and M. Fuller. Experimental and numerical investigations on drag and torque characteristics of three-bladed Savonius wind turbine. *ASME 2009 International Mechanical Engineering Congress and Exposition*, 6:85–94, 2009.

[97] F. Moulkalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab*. Springer International Publishing; 1st ed., 2016.

[98] G. Muller, M. F. Jentsch, and E. Stoddart. Vertical axis resistance type wind turbines for use in buildings. *Renewable Energy*, 34(5):1407–1412, 2009.

[99] T. Munk, D. Kane, and D. Yebra. The effects of corrosion and fouling on the performance of ocean-going vessels: a naval architectural perspective. In C. Hellio and D. Yebra, editors, *Advances in Marine Antifouling Coatings and Technologies*, Woodhead Publishing Series in Metals and Surface Engineering, pages 148 – 176. Woodhead Publishing, 2009.

[100] OpenCFD Ltd. The Open Source CFD Toolbox User Guide 2.1.1. `https://www.openfoam.com/documentation/user-guide/`.

[101] A. Orlandi, M. Collu, S. Zanforlin, and A. Shires. 3D URANS analysis of a vertical-axis wind turbine in skewed flows. *Journal of Wind Engineering and Industrial Aerodynamics*, 147:77–84, 2015.

[102] R. L. Panton. *Incompressible flow*. John Wiley and Sons; 4th edition, 2013.

[103] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow*. CRC Press; 1st edition, 1980.

[104] S. B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[105] N. Qin, R. Howell, N. Durrani, K. Hamad, and T. Smith. Unsteady flow simulation and dynamic stall around vertical axis wind turbine blades. *Wind Engineering*, 35(4):511–527, 2011.

[106] R. G. Rajagopalan and J. B. Fanucci. Finite difference model for vertical axis wind turbines. *Journal of Propulsion and Power*, 1(6):432–436, 1985.

[107] B. E. Rapp. *Microfluidics: Modeling, Mechanics, and Mathematics*. Elsevier; 1st edition, 2016.

[108] A. Rezaeiha, I. Kalkman, and B. Blocken. CFD simulation of a vertical-axis wind turbine operating at a moderate tip speed ratio: Guidelines for minimum domain size and azimuthal increment. *Renewable energy*, 107(C):373–385, 2017.

[109] U. K. Saha, S. Thotla, and D. Maity. Optimum design configuration of Savonius rotor through wind tunnel experiments. *Journal of Wind Engineering and Industrial Aerodynamics*, 96(8-9):1359–1375, 2008.

[110] M. Schäfer. *Computational Engineering: Introduction to Numerical Methods*. Springer; 2006 edition, 2006.

[111] H. Schlichting and K. Gersten. *Boundary-Layer Theory*. Springer-Verlag Berlin Heidelberg; 9th edition, 2017.

[112] F. G. Schmitt. About Boussinesq's turbulent viscosity hypothesis: historical remarks and a direct evaluation of its validity. *Comptes Rendus Mécanique*, 335(9–10):617–627, 2007.

[113] M. P. Schultz and G. W. Swain. The influence of biofilms on skin friction drag. *Biofouling*, 15:129–139, 2000.

[114] A. Shires and V. Kourkoulis. Application of circulation controlled blades for vertical axis wind turbines. *Energies*, 6(8):3744–3763, 2013.

[115] L. Shui, J. C. Eijkel, and A. van den Berg. Multiphase flow in microfluidic systems: Control and applications of droplets and interfaces. *Advances in Colloid and Interface Science*, 133(1):35–49, 2007.

[116] D. A. Spera. *Wind Turbine Technology: Fundamental Concepts of Wind Turbine Engineering*. American Society of Mechanical Engineers; 2nd edition, Jan 1 1994.

[117] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1-2):281–309, 2001.

[118] H. Sturm, G. Dumstorf, P. Busche, D. Westermann, and W. Lang. Boundary layer separation and reattachment detection on airfoils by thermal flow sensors. *Sensors*, 12:14292–14306, 2012.

[119] Y. Tahara. *A Reynolds-Averaged Navier–Stokes Equation Solver for Predicting Ship Viscous Flow with Free Surface Effects,Technical Notes on NAPA RaNS CFD Module*. Osaka Prefecture University, 2008.

[120] H. Tennekes and J. L. Lumley. *A first course in turbulence*. The MIT Press; 1st edition, 1972.

[121] J. Tong and A. Schwab. The Flight of Birds. `https://ocw.mit.edu/courses/materials-science-and-engineering/3-a26-freshman-seminar-the-nature-of-engineering-fall-2005/projects/flght_of_brdv2ed.pdf`, 2005. Online; accessed 8 March 2019.

[122] W. Tong. *Wind Power Generation and Wind Turbine Design*. WIT Press/Computational Mechanics; 1st edition, 2010.

[123] A. Toselli and O. B. Widlund. *Domain Decomposition Methods: Algorithms and Theory*. Springer-Verlag Berlin Heidelberg, 2005.

[124] K. Tota-Maharaj, R. Ramkissoon, and K. Manohar. Economical Darrieus straight bladed vertical axis wind turbine for renewable energy applications. *Journal of the Energy Institute*, 85(3):156–162, 2012.

[125] U.S. Government Department of Energy. U.S. Department of Energy on Green House Gases. `http://www.eia.doe.gov/bookshelf/brochures/greenhouse/Chapter1.htm2009`. Online; accessed 5 July 2017.

[126] V. Friesen. A technical report on simulating the Lux Wind Turbine. `https://simcity.usask.ca/trac/browser/LuxWindPower/papers/VaughnFriesenCMPT400/lux-turbine-simulation.pdf`, 2016.

[127] T. D. Valentine. *Reynolds-Averaged Navier–Stokes Codes and Marine Propulsor Analysis-Hydromechanics Directorate Research and Development Report*. Carderock Division, Naval Surface Warfare Center Bethesda, MD 20084-5000, 1993.

[128] S. P. Vanka. Block-implicit multigrid solution of Navier–Stokes equations in primitive variables. *J. Comp. Physics*, 65(1):138–158, 1986.

[129] P. S. Veers and S. R. Winterstein. Application of measured loads to wind turbine fatigue and reliability analysis. *Journal of Solar Energy Engineering*, 120(4):233–239, 1998.

[130] H. K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Pearson; 2nd edition, 2007.

[131] W. Bing-Chen. Fundamentals of Turbulence and Boundary-Layer Theory. `https://home.cc.umanitoba.ca/~wang44/Courses/MECH3492/Handout_Ch3.pdf`, 2017. Online; accessed 9 March 2019.

[132] Z. J. Wang. Spectral (finite) volume method for conservation laws on unstructured grids: Basic formulation. *Journal of Computational Physics*, 178(1):210–251, 2002.

[133] D. C. Wilcox. *Turbulence Modelling for CFD*. 3rd ed., DCW Industries, 2006.

[134] Z. Yang and T. H. Shih. New time scale based k-epsilon model for near-wall turbulence. *AIAA Journal*, 31(7):1191–1198, 1993.

[135] M. Zamani, M. J. Maghrebi, and S. R. Varedi. Starting torque improvement using J-shaped straight-bladed Darrieus vertical axis wind turbine by means of numerical simulation. *Renewable Energy*, 95(Supplement C):109–126, 2016.

[136] B. Zhang and C. C. Liang. A simple, efficient, high-order accurate sliding-mesh interface approach to the spectral difference method on coupled rotating and stationary domains. *Journal of Computational Physics*, 295(C):147–160, 2015.

Appendix A

Time-stepping schemes and meshing dictionary for the Lux VAWT simulation in OpenFOAM

BDF2 SCHEME in OpenFOAM

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
format      ascii;
class       dictionary;
location    "system";
object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
default  backward;// CrankNicolson 0.5858;
// CrankNicolson is transient
}

gradSchemes
{
default         Gauss linear;
grad(p)         Gauss linear;
grad(U)         cellLimited Gauss linear 1;
}

divSchemes
{
default         none;
div(phi,U)       Gauss linearUpwind grad(U);// i added the bounded term
div(phi,k)       Gauss upwind;
div(phi,omega)   Gauss upwind;
div((nuEff*dev2(T(grad(U)))))  Gauss linear;
}

laplacianSchemes
{
default         Gauss linear limited corrected 0.5;
}
```

```
interpolationSchemes
{
default         linear;
}

snGradSchemes
{
default   limited corrected 0.33;
// i can use 0.5 for better greater accuracy. 0.33 offers greater stability.
}

fluxRequired
{
default         no;
pcorr           ;
p                   ;
}

wallDist
{
method meshWave;
nRequired false;
}

// ************************************************************************* //
```

TR–BDF2 scheme implementation in OpenFOAM

```
/*---------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
License
This file is part of OpenFOAM.

OpenFOAM is free software: you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License
along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "TRRBDFDdtScheme.H"
#include "surfaceInterpolate.H"
```

```
#include "fvcDiv.H"
#include "fvMatrices.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace fv
{

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

template<class Type>
scalar TRRBDFDdtScheme<Type>::deltaT_() const
{
return mesh().time().deltaTValue();
}


template<class Type>
scalar TRRBDFDdtScheme<Type>::deltaT0_() const
{
return mesh().time().deltaT0Value();
}


template<class Type>
template<class GeoField>
scalar TRRBDFDdtScheme<Type>::deltaT0_(const GeoField& vf) const
{
if (vf.nOldTimes() < 2)
{
return GREAT;
}
else
{
return deltaT0_();
}
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

template<class Type>
tmp<GeometricField<Type, fvPatchField, volMesh> >
TRRBDFDdtScheme<Type>::fvcDdt
(
const dimensioned<Type>& dt
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();
```

```
IOobject ddtIOobject
(
"ddt("+dt.name()+')',
mesh().time().timeName(),
mesh()
);

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_();
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

if (mesh().moving())
{
tmp<GeometricField<Type, fvPatchField, volMesh> > tdtdt
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
mesh(),
dimensioned<Type>
(
"0",
dt.dimensions()/dimTime,
pTraits<Type>::zero
)
)
);

tdtdt().internalField() = rDeltaT.value()*dt.value()*
(
1.0 - (A1*mesh().V0())/mesh().V()
);

return tdtdt;
}
else
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
mesh(),
dimensioned<Type>
(
"0",
dt.dimensions()/dimTime,
pTraits<Type>::zero
),
calculatedFvPatchField<Type>::typeName
)
);
```

```
}
}


template<class Type>
tmp<GeometricField<Type, fvPatchField, volMesh> >
TRRBDFDdtScheme<Type>::fvcDdt
(
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();

IOobject ddtIOobject
(
"ddt("+vf.name()+')',
mesh().time().timeName(),
mesh()
);

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

if (mesh().moving())
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
mesh(),
rDeltaT.dimensions()*vf.dimensions(),
rDeltaT.value()*
(
vf.internalField() -
(
A1*vf.oldTime().internalField()*mesh().V0())/mesh().V()
),
rDeltaT.value()*
(
vf.boundaryField() -
(
A1*vf.oldTime().boundaryField()
)
)
)
);
}
else
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
```

```
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
rDeltaT*
(
vf
- A1*vf.oldTime()
)
)
);
}
}


template<class Type>
tmp<GeometricField<Type, fvPatchField, volMesh> >
TRRBDFDdtScheme<Type>::fvcDdt
(
const dimensionedScalar& rho,
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();

IOobject ddtIOobject
(
"ddt("+rho.name()+','+vf.name()+')',
mesh().time().timeName(),
mesh()
);

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

if (mesh().moving())
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
mesh(),
rDeltaT.dimensions()*rho.dimensions()*vf.dimensions(),
rDeltaT.value()*rho.value()*
(
vf.internalField() -
(
A1*vf.oldTime().internalField()*mesh().V0())/mesh().V()
),
rDeltaT.value()*rho.value()*
(
vf.boundaryField() -
```

```
(
A1*vf.oldTime().boundaryField()
)
)
)
);
}
else
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
rDeltaT*rho*
(
vf
- A1*vf.oldTime()
)
)
);
}
}


template<class Type>
tmp<GeometricField<Type, fvPatchField, volMesh> >
TRRBDFDdtScheme<Type>::fvcDdt
(
const volScalarField& rho,
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();

IOobject ddtIOobject
(
"ddt("+rho.name()+','+vf.name()+')',
mesh().time().timeName(),
mesh()
);

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

if (mesh().moving())
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
```

```
mesh(),
rDeltaT.dimensions()*rho.dimensions()*vf.dimensions(),
rDeltaT.value()*
(
rho.internalField()*vf.internalField() -
(
A1*rho.oldTime().internalField()
*vf.oldTime().internalField()*mesh().V0())/mesh().V()
),
rDeltaT.value()*
(
rho.boundaryField()*vf.boundaryField() -
(
A1*rho.oldTime().boundaryField()
*vf.oldTime().boundaryField()
)
)
)
);
}
else
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
rDeltaT*
(
rho*vf
- A1*rho.oldTime()*vf.oldTime()
)
)
);
}
}


template<class Type>
tmp<GeometricField<Type, fvPatchField, volMesh> >
TRRBDFDdtScheme<Type>::fvcDdt
(
const volScalarField& alpha,
const volScalarField& rho,
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();

IOobject ddtIOobject
(
"ddt("+alpha.name()+','+rho.name()+','+vf.name()+')',
mesh().time().timeName(),
mesh()
```

```
);

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

if (mesh().moving())
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
mesh(),
rDeltaT.dimensions()
*alpha.dimensions()*rho.dimensions()*vf.dimensions(),
rDeltaT.value()*
(

alpha.internalField()
*rho.internalField()
*vf.internalField() -
(
A1
*alpha.oldTime().internalField()
*rho.oldTime().internalField()
*vf.oldTime().internalField()*mesh().V0())/mesh().V()
),
rDeltaT.value()*
(

alpha.boundaryField()
*rho.boundaryField()
*vf.boundaryField() -
(
A1
*alpha.oldTime().boundaryField()
*rho.oldTime().boundaryField()
*vf.oldTime().boundaryField()
)
)
)
);
}
else
{
return tmp<GeometricField<Type, fvPatchField, volMesh> >
(
new GeometricField<Type, fvPatchField, volMesh>
(
ddtIOobject,
rDeltaT*
(
```

```
alpha*rho*vf
- A1*alpha.oldTime()*rho.oldTime()*vf.oldTime()
)
)
);
}
}


template<class Type>
tmp<fvMatrix<Type> >
TRRBDFDdtScheme<Type>::fvmDdt
(
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
tmp<fvMatrix<Type> > tfvm
(
new fvMatrix<Type>
(
vf,
vf.dimensions()*dimVol/dimTime
)
);

fvMatrix<Type>& fvm = tfvm();

scalar rDeltaT = 1.0/deltaT_();

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

fvm.diag() = (rDeltaT)*mesh().V();

if (mesh().moving())
{
fvm.source() = rDeltaT*
(
A1*vf.oldTime().internalField()*mesh().V0()

);
}
else
{
fvm.source() = rDeltaT*mesh().V()*
(
A1*vf.oldTime().internalField()

);
}

return tfvm;
```

```
}


template<class Type>
tmp<fvMatrix<Type> >
TRRBDFDdtScheme<Type>::fvmDdt
(
const dimensionedScalar& rho,
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
tmp<fvMatrix<Type> > tfvm
(
new fvMatrix<Type>
(
vf,
rho.dimensions()*vf.dimensions()*dimVol/dimTime
)
);
fvMatrix<Type>& fvm = tfvm();

scalar rDeltaT = 1.0/deltaT_();

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

fvm.diag() = (rDeltaT*rho.value())*mesh().V();

if (mesh().moving())
{
fvm.source() = rDeltaT*rho.value()*
(
A1*vf.oldTime().internalField()*mesh().V0()
);
}
else
{
fvm.source() = rDeltaT*mesh().V()*rho.value()*
(
A1*vf.oldTime().internalField()
);
}

return tfvm;
}


template<class Type>
tmp<fvMatrix<Type> >
TRRBDFDdtScheme<Type>::fvmDdt
(
const volScalarField& rho,
```

```
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
tmp<fvMatrix<Type> > tfvm
(
new fvMatrix<Type>
(
vf,
rho.dimensions()*vf.dimensions()*dimVol/dimTime
)
);
fvMatrix<Type>& fvm = tfvm();

scalar rDeltaT = 1.0/deltaT_();

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

fvm.diag() = (rDeltaT)*rho.internalField()*mesh().V();

if (mesh().moving())
{
fvm.source() = rDeltaT*
(
A1*rho.oldTime().internalField()
*vf.oldTime().internalField()*mesh().V0()
);
}
else
{
fvm.source() = rDeltaT*mesh().V()*
(
A1*rho.oldTime().internalField()
*vf.oldTime().internalField()
);
}

return tfvm;
}


template<class Type>
tmp<fvMatrix<Type> >
TRRBDFDdtScheme<Type>::fvmDdt
(
const volScalarField& alpha,
const volScalarField& rho,
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
tmp<fvMatrix<Type> > tfvm
(
```

```
new fvMatrix<Type>
(
vf,
alpha.dimensions()*rho.dimensions()*vf.dimensions()*dimVol/dimTime
)
);
fvMatrix<Type>& fvm = tfvm();

scalar rDeltaT = 1.0/deltaT_();

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

fvm.diag() =
(rDeltaT)*alpha.internalField()*rho.internalField()*mesh().V();

if (mesh().moving())
{
fvm.source() = rDeltaT*
(
A1
*alpha.oldTime().internalField()
*rho.oldTime().internalField()
*vf.oldTime().internalField()*mesh().V0()
);
}
else
{
fvm.source() = rDeltaT*mesh().V()*
(
A1
*alpha.oldTime().internalField()
*rho.oldTime().internalField()
*vf.oldTime().internalField()

);
}

return tfvm;
}


template<class Type>
tmp<typename TRRBDFDdtScheme<Type>::fluxFieldType>
TRRBDFDdtScheme<Type>::fvcDdtUfCorr
(
const GeometricField<Type, fvPatchField, volMesh>& U,
const GeometricField<Type, fvsPatchField, surfaceMesh>& Uf
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();
```

```
scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(U);
scalar gamma = 0.5858;
scalar A1 =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

return tmp<fluxFieldType>
(
new fluxFieldType
(
IOobject
(
"ddtCorr(" + U.name() + ',' + Uf.name() + ')',
mesh().time().timeName(),
mesh()
),
this->fvcDdtPhiCoeff(U.oldTime(), (mesh().Sf() & Uf.oldTime()))
*rDeltaT
*(
mesh().Sf()
& (
(A1*Uf.oldTime())
- fvc::interpolate
(
A1*U.oldTime()
)
)
)
)
);
}


template<class Type>
tmp<typename TRRBDFDdtScheme<Type>::fluxFieldType>
TRRBDFDdtScheme<Type>::fvcDdtPhiCorr
(
const GeometricField<Type, fvPatchField, volMesh>& U,
const fluxFieldType& phi
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(U);
scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

return tmp<fluxFieldType>
(
new fluxFieldType
(
IOobject
(
"ddtCorr(" + U.name() + ',' + phi.name() + ')',
```

```
mesh().time().timeName(),
mesh()
),
this->fvcDdtPhiCoeff(U.oldTime(), phi.oldTime())
*rDeltaT
*(
(A1*phi.oldTime())
- (
mesh().Sf()
& fvc::interpolate
(
A1*U.oldTime()
)
)
)
)
);
}


template<class Type>
tmp<typename TRRBDFDdtScheme<Type>::fluxFieldType>
TRRBDFDdtScheme<Type>::fvcDdtUfCorr
(
const volScalarField& rho,
const GeometricField<Type, fvPatchField, volMesh>& U,
const GeometricField<Type, fvsPatchField, surfaceMesh>& Uf
)
{
if
(
U.dimensions() == dimVelocity
&& Uf.dimensions() == rho.dimensions()*dimVelocity
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(U);

scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

GeometricField<Type, fvPatchField, volMesh> rhoU0
(
rho.oldTime()*U.oldTime()
);

GeometricField<Type, fvPatchField, volMesh> rhoU00
(
rho.oldTime().oldTime()*U.oldTime().oldTime()
);

return tmp<fluxFieldType>
```

```
(
new fluxFieldType
(
IOobject
(
"ddtCorr("
+ rho.name() + ',' + U.name() + ',' + Uf.name() + ')',
mesh().time().timeName(),
mesh()
),
this->fvcDdtPhiCoeff(rhoU0, mesh().Sf() & Uf.oldTime())
*rDeltaT
*(
mesh().Sf()
& (
(A1*Uf.oldTime())
- fvc::interpolate(A1*rhoU0)
)
)
)
);
}
else if
(
U.dimensions() == rho.dimensions()*dimVelocity
&& Uf.dimensions() == rho.dimensions()*dimVelocity
)
{
return fvcDdtUfCorr(U, Uf);
}
else
{
FatalErrorIn
(
"TRRBDFDdtScheme<Type>::fvcDdtPhiCorr"
)   << "dimensions␣of␣phi␣are␣not␣correct"
<< abort(FatalError);

return fluxFieldType::null();
}
}


template<class Type>
tmp<typename TRRBDFDdtScheme<Type>::fluxFieldType>
TRRBDFDdtScheme<Type>::fvcDdtPhiCorr
(
const volScalarField& rho,
const GeometricField<Type, fvPatchField, volMesh>& U,
const fluxFieldType& phi
)
{
if
(
```

```
U.dimensions() == dimVelocity
&& phi.dimensions() == rho.dimensions()*dimVelocity*dimArea
)
{
dimensionedScalar rDeltaT = 1.0/mesh().time().deltaT();

scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(U);

scalar gamma = 0.5858;
scalar A1   =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

GeometricField<Type, fvPatchField, volMesh> rhoU0
(
rho.oldTime()*U.oldTime()
);

GeometricField<Type, fvPatchField, volMesh> rhoU00
(
rho.oldTime().oldTime()*U.oldTime().oldTime()
);

return tmp<fluxFieldType>
(
new fluxFieldType
(
IOobject
(
"ddtCorr("
+ rho.name() + ',' + U.name() + ',' + phi.name() + ')',
mesh().time().timeName(),
mesh()
),
this->fvcDdtPhiCoeff(rhoU0, phi.oldTime())
*rDeltaT
*(
(A1*phi.oldTime())
- (
mesh().Sf()
& fvc::interpolate(A1*rhoU0)
)
)
)
);
}
else if
(
U.dimensions() == rho.dimensions()*dimVelocity
&& phi.dimensions() == rho.dimensions()*dimVelocity*dimArea
)
{
return fvcDdtPhiCorr(U, phi);
}
else
```

```
{
FatalErrorIn
(
"TRRBDFDdtScheme<Type>::fvcDdtPhiCorr"
)    << "dimensions␣of␣phi␣are␣not␣correct"
<< abort(FatalError);

return fluxFieldType::null();
}
}


template<class Type>
tmp<surfaceScalarField> TRRBDFDdtScheme<Type>::meshPhi
(
const GeometricField<Type, fvPatchField, volMesh>& vf
)
{
scalar deltaT = deltaT_();
scalar deltaT0 = deltaT0_(vf);

scalar gamma = 0.5858;
scalar A1    =(1.0-((1.0-gamma)*(1.0-gamma)))/(gamma*(2.0-gamma));

return tmp<surfaceScalarField>
(
new surfaceScalarField
(
IOobject
(
mesh().phi().name(),
mesh().time().timeName(),
mesh(),
IOobject::NO_READ,
IOobject::NO_WRITE,
false
),
A1*mesh().phi()
//- A1*mesh().phi().oldTime()
)
);
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace fv

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam

// ************************************************************************* //
```

BlockMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
format      ascii;
class       dictionary;
object      blockMeshDict;
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 1;

vertices
(

(-70 -100 -80)//0
(140 -100 -80)//1
(140  100 -80)//2
(-70  100 -80)//3
(-70 -100 80)//4
(140 -100 80)//5
(140  100 80)//6
(-70  100  80)//7
);
blocks
(
hex (0 1 2 3 4 5 6 7) (75 70 55) simpleGrading (1 1 1)
);

edges
(
);
boundary
(

inlet
{
type patch;
faces
(
(0 4 7 3)
);
}
outlet
{
```

```
type patch;
faces
(
(2 6 5 1)
);
}
topwall
{
type  patch;
faces
(
(3 7 6 2)
);
}
bottomwall
{
type wall;
faces
(
(1 5 4 0)
);
}

sideBack
{
type patch;
faces
(
(4 5 6 7)
);
}

sideFront
{
type patch;
faces
(
(0 3 2 1)
);
}
);
// ************************************************************************* //
```

SnappyHexMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
        version     2.0;
```

```
        format      ascii;
        class       dictionary;
        object      snappyHexMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

// Which of the steps to run
castellatedMesh true;// was set true before
snap            true;
addLayers       false;// was set false before

// set addLayers at first. you can always
// go back and change to true when there is no more error.
// Geometry. Definition of all surfaces. All surfaces are of class
// searchableSurface.
// Surfaces are used
// - to specify refinement for any mesh cell intersecting it
// - to specify refinement for any mesh cell inside/outside/near
// - to 'snap' the mesh boundary to the surface
geometry
{
        //M1outercubeM.stl {type triSurfaceMesh; name outercube;}
        //innercubeM.stl {type triSurfaceMesh; name innercube;}
        //innercylindersmal1.stl {type triSurfaceMesh; name innercylindersmal1;}
        //blades.stl {type triSurfaceMesh; name blades;}

        M1outercubeM.stl
        {
                type        triSurfaceMesh;
                name        M1outercubeM;
        }

        innercubeM.stl
        {
                type        triSurfaceMesh;
                name        innercubeM;
        }

        innercylindersmal1.stl
        {
                type        triSurfaceMesh;
                name        innercylindersmal1;
        }

        blades.stl
        {
                type        triSurfaceMesh;
                name        blades;
        }
        /* outercube.stl
        {
                type        triSurfaceMesh;
                name        outercube;
                regions
```

```
                {
                        outercube
                        {
                                name       outercube;
                        }
                }
        }

        innercube.stl
        {
                type        triSurfaceMesh;
                name        innercube;
                regions
                {
                        innercube
                        {
                                name       innercube;
                        }
                }
        }
        innercylindersmal1.stl
        {
                type        triSurfaceMesh;
                name        innercylindersmal1;
                regions
                {
                        innercylindersmal1
                        {
                                name       innercylindersmal1;

                        }
                }
        }

        blades.stl
        {
                type        triSurfaceMesh;
                name        blades;
                regions
                {
                        blades
                        {
                                name       blades;
                        }
                }
        }
*/      //refinementBox {type searchableBox; min (-25 -30.2509 -20.9932);
max ( 30 30.129 24.0068);
        };

        // Settings for the castellatedMesh generation.

        castellatedMeshControls
        {
```

```
        maxLocalCells 1500000; //was 1000000
        maxGlobalCells 20000000;
        minRefinementCells 0;
        maxLoadUnbalance 0.10;
        nCellsBetweenLevels 2;//

        // Explicit feature edge refinement
        // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        features
        //  taken from STL from each .eMesh file created
by "SurfaceFeatureExtract" command
        (
        {
                file        "innercylindersmal1.eMesh";
                level       5;
        }
        {
                file        "M1outercubeM.eMesh";
                level       0;
        }
        {
                file        "blades.eMesh";
                level       9; // was 6
        }
        //{
                //  file        "innercube.eMesh"; was uncommented initially
                //level       2; // was 2
                //}

        );

        // Surface based refinement
        // ~~~~~~~~~~~~~~~~~~~~~~~~

        refinementSurfaces
        {
                innercylindersmal1
                {
                        level       (5 5);

                        faceType    baffle; // was initially background
                        cellZone    innercylindersmal1;
                        faceZone    innercylindersmal1;
                        cellZoneInside  inside;
                }
                M1outercubeM
                {
                        level       (0 0);
                }
                blades
                {
                        level       (8 8);// was (8 8)
```

```
                }


        }


        resolveFeatureAngle 60;// was 30

        // Region-wise refinement
        // ~~~~~~~~~~~~~~~~~~~~~~

        refinementRegions
        {
                //innercube
                // {
                        //   mode        inside;
                        // levels      ((1E15 3));
                        // }
                innercylindersmal1
                {
                        mode        inside;
                        levels      ((1E15 3));
                }

        }

        // Mesh selection
        // ~~~~~~~~~~~~~~
locationInMesh (8.2 12.906 9.16859);
// use (8.2 12.906 9.16859) for setB and (8.25 14 0.00939633) for setBB
allowFreeStandingZoneFaces true; // was true
}

// Settings for the snapping.
snapControls
{

        nSmoothPatch 15; // was 5
        tolerance 2;
        nSolveIter 300;
        nRelaxIter 10; // was 10

        // Feature snapping
        nFeatureSnapIter 5;// was 5
        implicitFeatureSnap false;
        explicitFeatureSnap true;
        multiRegionFeatureSnap false;
//it was true before.
//Just for explicit method, "...detect features between multiple surfaces..."
}
addLayersControls
{

        relativeSizes true;// was false before
```

```
        layers

        {
                blades
                {
                        nSurfaceLayers 3;
                }

        }

        // Expansion factor for layer mesh
        expansionRatio 1.2;// was 1.0
        finalLayerThickness 0.1;
        minThickness 0.001;
        nGrow 0; // was 1
        // Advanced settings
        featureAngle 30; // was 80
        nRelaxIter 3;
        nSmoothSurfaceNormals 1;
        nSmoothNormals 3;
        nSmoothThickness 10;
        maxFaceThicknessRatio 0.5;
        maxThicknessToMedialRatio 0.3;
        minMedianAxisAngle 90;
        nBufferCellsNoExtrude 0;
        nLayerIter 50;
        slipFeatureAngle 30;
    }

    // Generic mesh quality settings. At any undoable phase these determine
    // where to undo.
    meshQualityControls
    {
        // Specify mesh quality constraints in separate dictionary so can
        // be reused (e.g. checkMesh -meshQuality)
        #include "meshQualityDict"
        relaxed
        {

                maxNonOrtho 75;
        }


        // Advanced
        nSmoothScale 4;
        errorReduction 0.75;
    }
    // Advanced
    // put 1 if you wnat to write all the meshes
//at each stage so that you look and make correction and 0 if not
    debug 0; //0: only write final meshes
    // debug 1;
    // Merge tolerance. Is fraction of overall bounding box of initial mesh.
    // Note: the write tolerance needs to be higher than this.
```

```
        mergeTolerance 1e-6;
        // ************************************************************************* //
```

checkMeshDict

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
Build  : 3.0.x-221b8ab77307
Exec   : checkMesh
Date   : Jul 10 2018
Time   : 14:46:49
Host   : "gottlieb"
PID    : 19670
Case   : /home/afagbade/storage/meshing
nProcs : 1
sigFpe : Enabling floating point exception trapping (FOAM_SIGFPE).
fileModificationChecking : Monitoring run-time modified files using timeStampMaster
allowSystemOperations : Allowing user-supplied system call operations

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
Create time

Create polyMesh for time = 0

Time = 0

Mesh stats
points:           9392566
faces:            24689365
internal faces:   22811389
cells:            7502065
faces per cell:   6.2761189
boundary patches: 9
point zones:      0
face zones:       1
cell zones:       1

Overall number of cells of each type:
hexahedra:     5957692
prisms:        96311
wedges:        0
pyramids:      0
tet wedges:    640
tetrahedra:    4
```

```
polyhedra:     1396233
Breakdown of polyhedra by number of faces:
faces   number of cells
4   166583
5   115640
6   389526
7   1041
8   1430
9   414430
10   1032
11   1180
12   222199
13   516
14   870
15   80142
16   28
17   56
18   1560

Checking topology...
Boundary definition OK.
Cell to face addressing OK.
Point usage OK.
Upper triangular ordering OK.
Face vertices OK.
*Number of regions: 2
The mesh has multiple regions which are not connected by any face.
<<Writing region information to "0/cellToRegion"
<<Writing region 0 with 1849690 cells to cellSet region0
<<Writing region 1 with 5601190 cells to cellSet region1

Checking patch topology for multiply connected surfaces...
Patch    Faces   Points                  Surface topology
inlet      759      816  ok (non-closed singly connected)
outlet      759      816  ok (non-closed singly connected)
topwall     2244     2346  ok (non-closed singly connected)
bottomwall     2772     2925  ok (non-closed singly connected)
blades   669570   874686      ok (closed singly connected)
AMI1   599372   599374      ok (closed singly connected)
AMI2   599372   599374      ok (closed singly connected)
frontSide     1564     1656  ok (non-closed singly connected)
backSide     1564     1656  ok (non-closed singly connected)

Checking geometry...
Overall domain bounding box (-50 -18.194361 -46.345638) (140 49.80563 50.313278)
Mesh has 3 geometric (non-empty/wedge) directions (1 1 1)
Mesh has 3 solution (non-empty) directions (1 1 1)
Boundary openness (-4.5762753e-18 7.548234e-16 -2.7526468e-16) OK.
Max cell openness = 4.6338651e-16 OK.
Max aspect ratio = 22.728766 OK.
Minimum face area = 4.8682578e-07. Maximum face area = 20.495095.
Face area magnitudes OK.
Min volume = 7.8079192e-09. Max volume = 48.830264.  Total volume = 1248810.1.
Cell volumes OK.
```

```
Mesh non-orthogonality Max: 65.001965 average: 13.664747
Non-orthogonality check OK.
Face pyramids OK.
Max skewness = 3.855896 OK.
Coupled point location match (average 0) OK.

Mesh OK.

End
```

Appendix B

Solver settings

For smooth and successful simulation of the Luxfoam case with the PimpleDyMFoam solver, the simulation parameters must be explicitly set in the `controlDict` of the system directory. The control settings such as start/end time of the simulation, fixed/variable time-step, and also the format of the simulation output may be outlined within the `controlDict` file as

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
format      ascii;
class       dictionary;
location    "system";
object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     pimpleDyMFoam;

startFrom       latestTime;

startTime       0;

stopAt          endTime;

endTime         24;

deltaT          0.002777;

writeControl    runTime;

writeInterval   0.01;
// you can specify just the amount of time for one complete revolution.

purgeWrite      0;// tell to write out results in separate directories (0)

writeFormat     ascii;// was binary

writePrecision  8;

writeCompression uncompressed;// was compressed

timeFormat      general;
```

```
timePrecision   16;

runTimeModifiable true;
// help you to make modification to the case while it is running

adjustTimeStep  no;//you can set yes if you want variable time step

maxCo           30;

libs
(
"libOpenFOAM.so"
"libforces.so"
"libincompressibleTurbulenceModels.so"

);

functions
{

forces
{
type                forces;
functionObjectLibs  ("libforces.so");
outputControl       timeStep;
outputInterval      1;
patches             (bladewalls);
pName               p;
UName               U;
rhoName             rhoInf;
log                 true;//
rhoInf              1.225;//1225 may be used< change to 1.225
CofR                (0 0 0);
pitchAxis           (0 1 0);//same as axis of rotation
}

#include "readFields"
#include "Q"
#include "surfaces"

}
// ************************************************************************* //
```

Also, the selection of finite volume discretization schemes for various derivatives terms in the Navier–Stokes equations is made in the `fvSchemes` dictionary within the system directory. A typical example of the setting is given as

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
```

```
{
version     2.0;
format      ascii;
class       dictionary;
location    "system";
object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
ddtSchemes
{
default               CrankNicolson 0.5858;
}
gradSchemes
{
default         Gauss linear;
grad(p)         Gauss linear;
grad(U)         cellLimited Gauss linear 1;
}
divSchemes
{
default         none;
div(phi,U)      Gauss linearUpwind grad(U);
div(phi,k)      Gauss upwind;
div(phi,omega)  Gauss upwind;
div((nuEff*dev2(T(grad(U))))) Gauss linear;
}
laplacianSchemes
{
default         Gauss linear limited corrected 0.5;
}

interpolationSchemes
{
default         linear;
}
snGradSchemes
{
default         limited corrected 0.33;
}

fluxRequired
{
default         no;
pcorr           ;
p                   ;
}

wallDist
{
method meshWave;
nRequired false;
}

// ************************************************************************* //
```

The discretization schemes for the derivative $\frac{\partial}{\partial t}$ are given in the `ddtSchemes` sub-dictionary. The `ddtSchemes` file consists of the implicit Euler scheme, the BDF2 scheme (otherwise called the backward scheme in OpenFOAM), the Crank–Nicolson scheme, and the newly implemented time-stepping scheme–TR–BDF2 (see Appendix A). These time-stepping schemes have been applied successfully in this simulation.

The `gradSchemes` sub-dictionary specifies the discretization scheme for the gradient terms of the discretized Navier–Stokes equations. As indicated, `Gauss linear` and `cellLimited Gauss linear` have been applied for p and U respectively. The keyword `Gauss` suggests that the standard finite volume integration, which requires the interpolation of values from the control volume center to the face center is applied. The `Gauss` scheme entry is followed by the choice of an interpolation scheme, which is linear. Sometimes different gradient limiters along with `Gauss` can be used to correct possible oscillation during computations. The `divSchemes` sub-dictionary handles the discretization of divergence terms. As observed, the `Gauss` scheme is chosen along with different interpolation schemes. For instance, for the divergence terms, upwind first-order bounded interpolation schemes have been chosen for the simulation. However, for the diffusive of the stress tensor, a linear interpolation scheme is applied. The `laplacianSchemes` sub-dictionary handles the discretization of the Laplacian terms in the discretized Navier–Stokes equations by the Gauss integration techniques along with unbounded, second-order conservative non-orthogonal correction Gauss linear scheme. The normal gradient term of the flow is discretized with the `snGradSchemes` using an explicit non-orthogonal correction scheme.

After the selection of both the spatial and time-stepping discretization scheme, the next computational approach is to choose a solver to handle the system of algebraic equations formed from the various discretization techniques. This step is controlled by the `fvSolution` dictionary in the system directory and illustrated as

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
format      ascii;
class       dictionary;
location    "system";
object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
pcorr
{
solver                  GAMG;
tolerance               0.01;
relTol                  0;
smoother                GaussSeidel;
nPreSweeps              0;
nPostSweeps             2;
cacheAgglomeration      true;
agglomerator            faceAreaPair;
nCellsInCoarsestLevel   50;
mergeLevels             1;
}
```

```
p
{
$pcorr;
tolerance       1e-6;
relTol          0.01;
}

pFinal
{
$p;
tolerance       1e-6;
relTol          0;
}

"(U|k|omega)"
{
solver          smoothSolver;
smoother        symGaussSeidel;
tolerance       1e-6;
relTol          0.1;
}

"(U|k|omega)Final"
{
solver          smoothSolver;
smoother        symGaussSeidel;
tolerance       1e-6;
relTol          0;
}

Phi
{
$p;
}
}

PIMPLE
{
correctPhi          no;
nOuterCorrectors    20;
nCorrectors         3;
nNonOrthogonalCorrectors 1;
pRefCell        1001;
pRefValue       0;

residualControl
{
U
{
tolerance 1e-2;
relTol 0;
}
```

```
p
{
tolerance 1e-2;
relTol 0;
}
}
}

relaxationFactors
{
fields
{
p               0.7;
}
equations
{
"(U|k|omega)"      0.8;
"(U|k|omega)Final" 1.0;
}
}

cache
{
grad(U);
}

// ************************************************************************* //
```

As indicated, the type of linear solvers to handle each flow variable of the discretized equations is specified. For instance, for the pressure discretized equation, GAMG (geometric-algebraic multi-grid) is used as the linear solver along with specified tolerance and smoother. For the transport variable U, k, and ω, smoothSolver has been applied as the linear solver. The `relaxationFactor` handles the under-relaxation of the solution. It works by controlling the amount by which a given variable changes from one iteration to the next.

OpenFOAM solver code to solve N-S equations. The code line from (a) to (b) implements PISO algorithm.

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

// define field vector fluid velocity u and f, face flux phi and pressure p
volVectorField u, f;
volScalarField p;
surfaceScalarField phi;
// define constant parameter fluid dynamical viscoity nu
scalarField nu;
// construct the fluid velocity equation
```

```
fvVectorMatrix UEqn(fvm::ddt(U)+fvm::div(phi, u)-fvm::laplacian (nu, u)-f)
(a)
//solve the momentum equation using explicit pressure
solve(UEqtn==-fvc::grad(p))
// predict the intermediate fluid velocity to calculate face flux
volVectorField rUA=1.0/UEqtn.A();
u=rUA*UEqtn.H();
phi=fvc::interpolate(u)& mesh.Sf();
//construct the pressure equation using the constraint from continuity equation
fvScalarMatrix pEqtn(
fvm::laplacian(rUA,p)==fvc::div(phi))
pEqtn.solve ();
//correct the fluid velocity by the post-solve pressure and update face flux
u=u-rUA*fvc::grad(p);
phi=phi-peqtn.flux();
(b)


// ************************************************************************* //
```

DynamicMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.3.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
format      ascii;
class       dictionary;
object      dynamicMeshDict;
}


dynamicFvMesh   solidBodyMotionFvMesh;

motionSolverLibs ("libfvMotionSolvers.so");

solidBodyMotionFvMeshCoeffs
{
cellZone        innercylindersmal1;
solidBodyMotionFunction  rotatingMotion;
rotatingMotionCoeffs
{
origin          (0 0 0);
axis            (0 1 0);
omega         -3.141526; // rad/s
}
}
```

```
// ************************************************************************* //
```

Physical properties

Defining both the chemical and physical properties of the working fluid (air) is necessary, besides the dynamic mesh and solver settings. Here, the physical properties of the flow such as the viscosity of air and the type of turbulence models can be specified in the **transportProperties** and **turbulenceProperties** files of the constant directory.

transportProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
format      ascii;
class       dictionary;
object      transportProperties;
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

transportModel Newtonian;

nu              nu [0 2 -1 0 0 0 0] 0.000001;


// ************************************************************************* //
```

The word nu (ν) actually denotes the kinematic viscosity of air with the default value of 0.000001 $m^2\,s^{-1}$.

For the **turbulenceProperties** file illustrated below, one can choose any of the three turbulence models for the simulation; the laminar, which does not require turbulence model, RASModel, which incorporates Reynolds-averaged simulation (RAS) modeling, and LESModel, which applies LES modeling.

TurbulenceProperties

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
```

```
format      ascii;
class       dictionary;
location    "constant";
object      turbulenceProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

simulationType  RAS;

RAS
{
#include "RASProperties"
}


// ************************************************************************* //
```

RASProperties

For the Luxfoam simulation case, the RASModel has been selected. The control setting for the RAS modeling is outlined in the **RASProperties** file of the constant directory.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.3.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
version     2.0;
format      ascii;
class       dictionary;
location    "constant";
object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

RASModel        kOmegaSST;

turbulence      on;

printCoeffs     on;


// ************************************************************************* //
```

The library of all available RAS models in OpenFOAM for incompressible fluids is summarized in Table (B.1). In the Luxfoam case, k-ω SST model [94] has been applied.

laminar	Dummy turbulence model for laminar flow
kEpsilon	Standard high-Re k–ϵ model
kOmega	Standard high -Re k–ω model
kOmegaSST	k–ω-SST model
RHGkEpsilon	RNG k–ϵ model
realizableKE	Realizable k–ϵ model
SpalartAllmaras	Spalart–Allmara 1-equation mixing-length model

Table B.1: RAS turbulence models for incompressible fluids–incompressibleRASModels [100]

Boundary Patches

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.3.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

FoamFile
{
version     2.0;
format      ascii;
class       polyBoundaryMesh;
location    "constant/polyMesh";
object      boundary;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
9
{
inlet
{
type            wall;
inGroup         1(wall);
nFaces          1878;
startFace       9900583;
}
outlet
{
type            patch;
nFaces          1878;
startFace       9902461;
}
bladewalls
{
type            wall;
inGroup         1(wall);
nFaces          14830;
startFace       9885753;
}
wall_rotation
{
type            wall;
inGroup         1(wall);
```

```
nFaces          7542;
startFace       9912421;
}
bottom_wall
{
type            wall;
inGroup         1(wall);
nFaces          7668;
startFace       9919963;
}
top_symmetry
{
type            symmetry;
inGroup         1(symmetry);
nFaces          8082;
startFace       9904339;
}
side_symmetry
{
type            symmetry;
inGroup         1(symmetry);
nFaces          8082;
startFace       9904339;
}
AMI1
{
type            cyclicAMI;
inGroup         1(cyclicAMI);
nFaces          71572;
startFace       9927631;
matchTolerance  0.0001;
transform       noOrdering;
neighbourPatch  AMI2;
}
AMI2
{
type            cyclicAMI;
inGroup         1(cyclicAMI);
nFaces          1756;
startFace       9999203;
matchTolerance  0.0001;
transform       noOrdering;
neighbourPatch  AMI1;
}
}
// ************************************************************************* //
```

Appendix C

Experimental Data from the Lux VAWT

Rotor Diameter		18.3 m
Rotor Height		29.3 m
Swept Area		360 m^2
Shaft Output Power		55 kW
Rotational Speed		Variable between 30 and 40 rpm
Cut in Wind Speed		4 m/s
Cut out Wind Speed		25 m/s
Generator		
	Direct Drive Rotation Speed	Variable between 30 and 40 rpm
	Gearbox Speed Increaser Ratio 45 to 1	Variable between 1350 and 1800 rpm
	Generator will need to supply	up to 10 kW to start the turbine
	Maximum Power Output	50 kW @ 40 rpm
Blades		
	Number	6
	Chord Length	200 mm (8 *in*)
	Material	Aluminum extruded 6063 T5
	Wall Thickness	32 mm (1/8 *in*)
	Number of Spars	1
Guy Cables		
	Number	3
	Diameter	13 mm (1/2 *in*)
	Tension	2270 kg (5,000 *lbs*)
	Angle to Horizontal	35 deg
	Length (each)	45 m
Cross Cables		
	Number	18
	Diameter	4–6 mm (5/32 in –1/4 *in*)
Blade Curvature		
	Radius	14.8 m
	Length	31.1 m

Table C.1: Lux Turbine Specification for 50 kW VAWT.

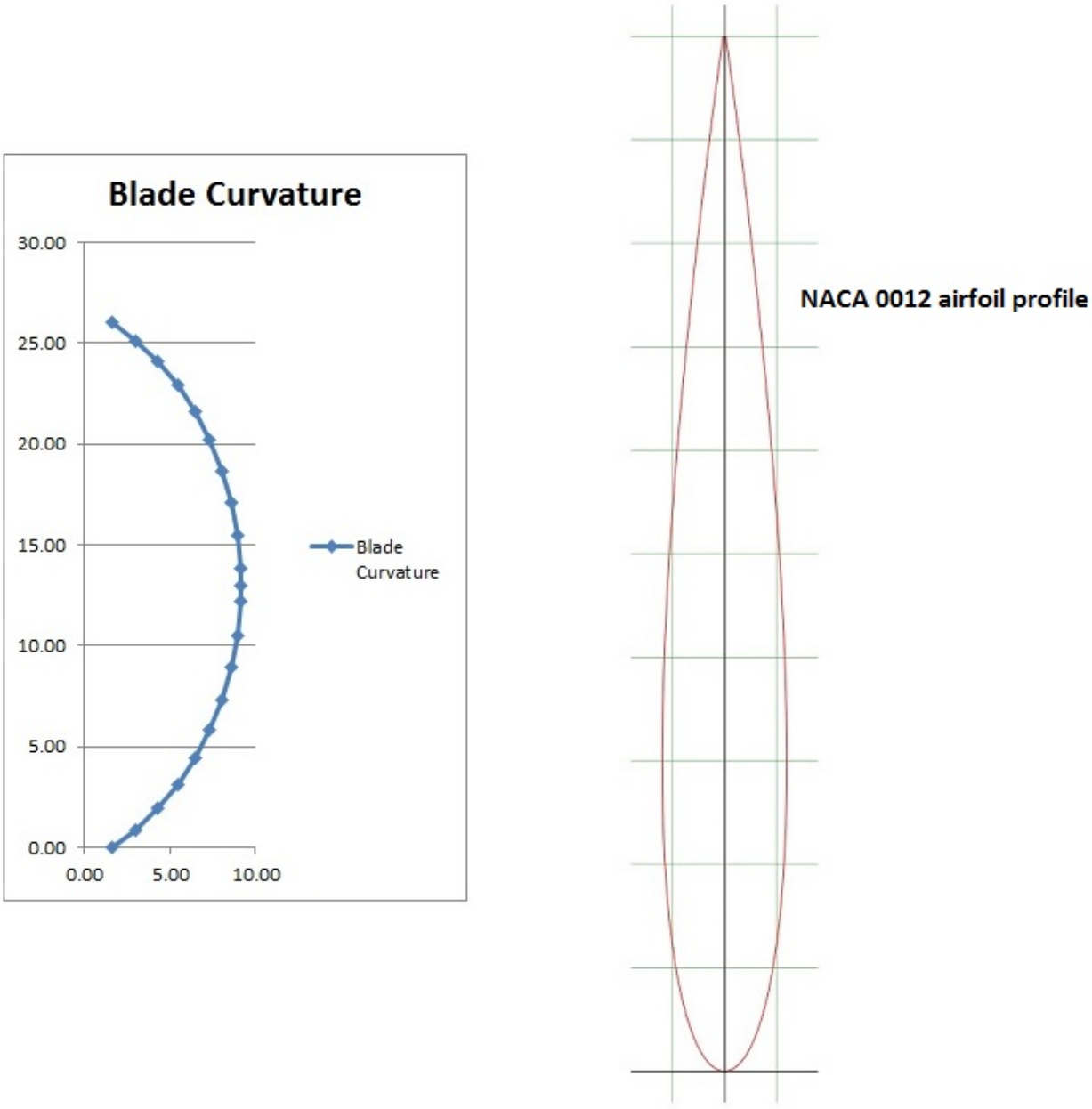


Figure C.1: The Lux blade curvature and the symmetrical NACA 0012 airfoil profile.

RPM	MPH	m/s	TSR	Cp	Hrs/Yr	Power (kW)	Total
30	8.95	4	7.19	0.18	582	2.6	1,513
34	11.19	5	6.52	0.25	667	7.0	4,666
38	13.42	6	6.07	0.28	719	13.5	9,706
38	15.66	7	5.20	0.29	739	21.8	16,120
40	17.90	8	4.79	0.28	731	31.9	23,308
40	20.13	9	4.26	0.25	697	40.8	28,441
40	22.37	10	3.83	0.21	644	47.2	30,407
40	24.61	11	3.48	0.17	578	49.3	28,500
40	26.85	12	3.19	0.12	505	47.6	24,018
40	29.08	13	2.95	0.09	429	45.1	19,346
40	31.32	14	2.74	0.07	356	42.9	15,253
40	33.56	15	2.56	0.06	288	41.4	11,905
40	35.79	16	2.40	0.04	227	40.6	9,221
40	38.03	17	2.25	0.04	175	40.3	7,062
40	40.27	18	2.13	0.03	132	40.1	5,299
40	42.51	19	2.02	0.03	97	40.2	3,917
40	44.74	20	1.92	0.02	70	40.8	2,867
40	46.98	21	1.83	0.02	50	41.8	2,072
40	49.22	22	1.74	0.02	34	42.9	1,469
						Total kWh/year	243,576

Table C.2: The Lux VAWT performance characteristics provided by Glen Lux

Appendix D

Force library for post-processing in OpenFOAM

```
/*---------------------------------------------------------------------------*\
=========                 |
\\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
 \\    /   O peration     |
  \\  /    A nd           | Copyright (C) 2011-2013 OpenFOAM Foundation
   \\/     M anipulation  |
-------------------------------------------------------------------------------
License
This file is part of OpenFOAM.

OpenFOAM is free software: you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License
along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Class
Foam::forces

Group
grpForcesFunctionObjects

Description
This function object calculates the forces and moments by integrating the
pressure and skin-friction forces over a given list of patches.

Member function forces::write() calculates the forces/moments and
writes the forces/moments into the file \<timeDir\>/forces.dat and bin
data (if selected) to the file \<timeDir\>/forces_bin.dat

Example of function object specification:
\verbatim
forces1
{
        type        forces;
        functionObjectLibs ("libforces.so");
        ...
        log         yes;
        patches     (walls);
```

```
        binData
        {
                nBin        20;
                direction   (1 0 0);
                cumulative  yes;
        }
}
\endverbatim

\heading Function object usage
\table
Property     | Description             | Required    | Default value
type         | type name: forces       | yes         |
log          | write force data to standard output | no | no
patches      | patches included in the forces calculation | yes |
pName        | pressure field name     | no          | p
UName        | velocity field name     | no          | U
rhoName      | density field name (see below) | no   | rho
CofR         | centre of rotation (see below) | no   |
directForceDensity | force density supplied directly (see below)|no|no
fDName       | name of force density field (see below) | no | fD
\endtable

Bin data is optional, but if the dictionary is present, the entries must
be defined according o
\table
nBin         | number of data bins     | yes         |
direction    | direction along which bins are defined | yes |
cumulative   | bin data accumulated with incresing distance | yes |
\endtable

Note
- For incompressible cases, set \c rhoName to \c rhoInf.  You will then be
required to provide a \c rhoInf value corresponding to the free-stream
constant density.
- If the force density is supplied directly, set the \c directForceDensity
flag to 'yes', and supply the force density field using the \c
fDName entry
- The centre of rotation (CofR) for moment calculations can either be
specified by an \c CofR entry, or be taken from origin of the local
coordinate system.  For example,
\verbatim
CofR        (0 0 0);
\endverbatim
or
\verbatim
coordinateSystem
{
        origin  (0 0 0);
        e3      (0 0 1);
        e1      (1 0 0);
}
\endverbatim
```

```
SeeAlso
Foam::functionObject
Foam::OutputFilterFunctionObject
Foam::forceCoeffs

SourceFiles
forces.C
IOforces.H

\*---------------------------------------------------------------------------*/

#ifndef forces_H
#define forces_H

#include "functionObjectFile.H"
#include "coordinateSystem.H"
#include "coordinateSystems.H"
#include "primitiveFieldsFwd.H"
#include "volFieldsFwd.H"
#include "HashSet.H"
#include "Tuple2.H"
#include "OFstream.H"
#include "Switch.H"
#include "writer.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

        // Forward declaration of classes
        class objectRegistry;
        class dictionary;
        class polyMesh;
        class mapPolyMesh;

/*---------------------------------------------------------------------------*\
Class forces Declaration
\*---------------------------------------------------------------------------*/

        class forces
        :
        public functionObjectFile
        {
                protected:

                // Protected data

                //- Name of this set of forces,
                //  Also used as the name of the probes directory.
                word name_;

                const objectRegistry& obr_;
```

```
//- On/off switch
bool active_;

//- Switch to send output to Info as well as to file
Switch log_;

//- Pressure, viscous and porous force per bin
List<Field<vector> > force_;

//- Pressure, viscous and porous moment per bin
List<Field<vector> > moment_;


// Read from dictionary

//- Patches to integrate forces over
labelHashSet patchSet_;

//- Name of pressure field
word pName_;

//- Name of velocity field
word UName_;

//- Name of density field (optional)
word rhoName_;

//- Is the force density being supplied directly?
Switch directForceDensity_;

//- The name of the force density (fD) field
word fDName_;

//- Reference density needed for incompressible calculations
scalar rhoRef_;

//- Reference pressure
scalar pRef_;

//- Coordinate system used when evaluting forces/moments
coordinateSystem coordSys_;

//- Flag to indicate whether we are using a local co-ordinate sys
bool localSystem_;

//- Flag to include porosity effects
bool porosity_;


// Bin information

//- Number of bins
label nBin_;
```

```
//- Direction used to determine bin orientation
vector binDir_;

//- Distance between bin divisions
scalar binDx_;

//- Minimum bin bounds
scalar binMin_;

//- Bin positions along binDir
List<point> binPoints_;

//- Should bin data be cumulative?
bool binCumulative_;


//- Initialised flag
bool initialised_;


// Protected Member Functions

//- Create file names for forces and bins
wordList createFileNames(const dictionary& dict) const;

//- Output file header information
virtual void writeFileHeader(const label i);

//- Initialise the fields
void initialise();

//- Return the effective viscous stress (laminar + turbulent).
tmp<volSymmTensorField> devRhoReff() const;

//- Dynamic viscosity field
tmp<volScalarField> mu() const;

//- Return rho if rhoName is specified otherwise rhoRef
tmp<volScalarField> rho() const;

//- Return rhoRef if the pressure field is dynamic, i.e. p/rho
//  otherwise return 1
scalar rho(const volScalarField& p) const;

//- Accumulate bin data
void applyBins
(
const vectorField& Md,
const vectorField& fN,
const vectorField& fT,
const vectorField& fP,
const vectorField& d
);
```

```
//- Helper function to write force data
void writeForces();

//- Helper function to write bin data
void writeBins();

//- Disallow default bitwise copy construct
forces(const forces&);

//- Disallow default bitwise assignment
void operator=(const forces&);


public:

//- Runtime type information
TypeName("forces");


// Constructors

//- Construct for given objectRegistry and dictionary.
//  Allow the possibility to load fields from files
forces
(
const word& name,
const objectRegistry&,
const dictionary&,
const bool loadFromFiles = false,
const bool readFields = true
);


//- Construct from components
forces
(
const word& name,
const objectRegistry&,
const labelHashSet& patchSet,
const word& pName,
const word& UName,
const word& rhoName,
const scalar rhoInf,
const scalar pRef,
const coordinateSystem& coordSys
);


//- Destructor
virtual ~forces();


// Member Functions
```

```
                //- Return name of the set of forces
                virtual const word& name() const
                {
                        return name_;
                }

                //- Read the forces data
                virtual void read(const dictionary&);

                //- Execute, currently does nothing
                virtual void execute();

                //- Execute at the final time-loop, currently does nothing
                virtual void end();

                //- Called when time was set at the end of the Time::operator++
                virtual void timeSet();

                //- Write the forces
                virtual void write();

                //- Calculate the forces and moments
                virtual void calcForcesMoment();

                //- Return the total force
                virtual vector forceEff() const;

                //- Return the total moment
                virtual vector momentEff() const;

                //- Update for changes of mesh
                virtual void updateMesh(const mapPolyMesh&)
                {}

                //- Update for changes of mesh
                virtual void movePoints(const polyMesh&)
                {}
        };


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

#endif

// ************************************************************************* //

/*--------------------------------------------------------------------------*\
=========                 |
\\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
```

```
\\    /   O peration     |
\\  /    A nd           | Copyright (C) 2011-2015 OpenFOAM Foundation
\\/     M anipulation  |
\*---------------------------------------------------------------------------*/
License
This file is part of OpenFOAM.

OpenFOAM is free software: you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License
along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "forces.H"
#include "volFields.H"
#include "dictionary.H"
#include "Time.H"
#include "wordReList.H"
#include "fvcGrad.H"
#include "porosityModel.H"
#include "turbulentTransportModel.H"
#include "turbulentFluidThermoModel.H"

// * * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * *//

namespace Foam
{
        defineTypeNameAndDebug(forces, 0);
}


// * * * * * * * * * * * * * * * Protected Member Functions  * * * * * * * *//

Foam::wordList Foam::forces::createFileNames(const dictionary& dict) const
{
        DynamicList<word> names(1);

        const word forceType(dict.lookup("type"));

        if (dict.found("binData"))
        {
                const dictionary& binDict(dict.subDict("binData"));
                label nb = readLabel(binDict.lookup("nBin"));
                if (nb > 0)
                {
```

```
                names.append(forceType + "_bins");
            }
        }

        names.append(forceType);

        return names;
}


void Foam::forces::writeFileHeader(const label i)
{
        if (i == 0)
        {
            // force data

            writeHeader(file(i), "Forces");
            writeHeaderValue(file(i), "CofR", coordSys_.origin());
            writeCommented(file(i), "Time");

            file(i)
            << "forces(pressure␣viscous␣porous)␣"
            << "moment(pressure␣viscous␣porous)";

            if (localSystem_)
            {
                file(i)
                << tab
                << "localForces(pressure,viscous,porous)␣"
                << "localMoments(pressure,viscous,porous)";
            }
        }
        else if (i == 1)
        {
            // bin data

            writeHeader(file(i), "Force␣bins");
            writeHeaderValue(file(i), "bins", nBin_);
            writeHeaderValue(file(i), "start", binMin_);
            writeHeaderValue(file(i), "delta", binDx_);
            writeHeaderValue(file(i), "direction", binDir_);

            vectorField binPoints(nBin_);
            writeCommented(file(i), "x␣co-ords␣␣:");
            forAll(binPoints, pointI)
            {
                binPoints[pointI] = (binMin_ + (pointI + 1)*binDx_)*binDir_;
                file(i) << tab << binPoints[pointI].x();
            }
            file(i) << nl;

            writeCommented(file(i), "y␣co-ords␣␣:");
            forAll(binPoints, pointI)
            {
```

```
                        file(i) << tab << binPoints[pointI].y();
                }
                file(i) << nl;

                writeCommented(file(i), "z␣co-ords␣␣:");
                forAll(binPoints, pointI)
                {
                        file(i) << tab << binPoints[pointI].z();
                }
                file(i) << nl;

                writeCommented(file(i), "Time");

                for (label j = 0; j < nBin_; j++)
                {
                        const word jn('(' + Foam::name(j) + ')');
                        const word f("forces" + jn + "[pressure,viscous,porous]");
                        const word m("moments" + jn + "[pressure,viscous,porous]");

                        file(i)<< tab << f << tab << m;
                }
                if (localSystem_)
                {
                        for (label j = 0; j < nBin_; j++)
                        {
                const word jn('(' + Foam::name(j) + ')');
                const word f("localForces" + jn + "[pressure,viscous,porous]");
                const word m("localMoments" + jn + "[pressure,viscous,porous]");

                file(i)<< tab << f << tab << m;
                        }
                }
        }
        else
        {
                FatalErrorIn("void␣Foam::forces::writeFileHeader(const␣label)")
                << "Unhandled␣file␣index:␣" << i
                << abort(FatalError);
        }

        file(i)<< endl;
}


void Foam::forces::initialise()
{
        if (initialised_ || !active_)
        {
                return;
        }

        if (directForceDensity_)
        {
                if (!obr_.foundObject<volVectorField>(fDName_))
```

```
                {
                        active_ = false;
                        WarningIn("void␣Foam::forces::initialise()")
                        << "Could␣not␣find␣" << fDName_ << "␣in␣database." << nl
                        << "␣␣␣␣De-activating␣forces."
                        << endl;
                }
        }
        else
        {
                if
                (
                !obr_.foundObject<volVectorField>(UName_)
                || !obr_.foundObject<volScalarField>(pName_)
                || (
                rhoName_ != "rhoInf"
                && !obr_.foundObject<volScalarField>(rhoName_)
                )
                )
                {
                        active_ = false;

                        WarningIn("void␣Foam::forces::initialise()")
                        << "Could␣not␣find␣" << UName_ << "␣," << pName_;

                        if (rhoName_ != "rhoInf")
                        {
                                Info<< "␣or␣" << rhoName_;
                        }

                        Info<< "␣in␣database." << nl
                        << "␣␣␣␣De-activating␣forces." << endl;
                }
        }

        initialised_ = true;
}


Foam::tmp<Foam::volSymmTensorField> Foam::forces::devRhoReff() const
{
        typedef compressible::turbulenceModel cmpTurbModel;
        typedef incompressible::turbulenceModel icoTurbModel;

        if (obr_.foundObject<cmpTurbModel>(cmpTurbModel::propertiesName))
        {
                const cmpTurbModel& turb =
                obr_.lookupObject<cmpTurbModel>(cmpTurbModel::propertiesName);

                return turb.devRhoReff();
        }
        else if (obr_.foundObject<icoTurbModel>(icoTurbModel::propertiesName))
        {
                const incompressible::turbulenceModel& turb =
```

```
            obr_.lookupObject<icoTurbModel>(icoTurbModel::propertiesName);

            return rho()*turb.devReff();
        }
        else if (obr_.foundObject<fluidThermo>(fluidThermo::typeName))
        {
            const fluidThermo& thermo =
            obr_.lookupObject<fluidThermo>(fluidThermo::typeName);

            const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);

            return -thermo.mu()*dev(twoSymm(fvc::grad(U)));
        }
        else if
        (
        obr_.foundObject<transportModel>("transportProperties")
        )
        {
            const transportModel& laminarT =
            obr_.lookupObject<transportModel>("transportProperties");

            const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);

            return -rho()*laminarT.nu()*dev(twoSymm(fvc::grad(U)));
        }
        else if (obr_.foundObject<dictionary>("transportProperties"))
        {
            const dictionary& transportProperties =
            obr_.lookupObject<dictionary>("transportProperties");

            dimensionedScalar nu(transportProperties.lookup("nu"));

            const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);

            return -rho()*nu*dev(twoSymm(fvc::grad(U)));
        }
        else
        {
            FatalErrorIn("forces::devRhoReff()")
            << "No valid model for viscous stress calculation"
            << exit(FatalError);

            return volSymmTensorField::null();
        }
}


Foam::tmp<Foam::volScalarField> Foam::forces::mu() const
{
        if (obr_.foundObject<fluidThermo>(basicThermo::dictName))
        {
            const fluidThermo& thermo =
            obr_.lookupObject<fluidThermo>(basicThermo::dictName);
```

```
            return thermo.mu();
        }
        else if
        (
        obr_.foundObject<transportModel>("transportProperties")
        )
        {
            const transportModel& laminarT =
            obr_.lookupObject<transportModel>("transportProperties");

            return rho()*laminarT.nu();
        }
        else if (obr_.foundObject<dictionary>("transportProperties"))
        {
            const dictionary& transportProperties =
            obr_.lookupObject<dictionary>("transportProperties");

            dimensionedScalar nu(transportProperties.lookup("nu"));

            return rho()*nu;
        }
        else
        {
            FatalErrorIn("forces::mu()")
            << "No valid model for dynamic viscosity calculation"
            << exit(FatalError);

            return volScalarField::null();
        }
}


Foam::tmp<Foam::volScalarField> Foam::forces::rho() const
{
        if (rhoName_ == "rhoInf")
        {
            const fvMesh& mesh = refCast<const fvMesh>(obr_);

            return tmp<volScalarField>
            (
            new volScalarField
            (
            IOobject
            (
            "rho",
            mesh.time().timeName(),
            mesh
            ),
            mesh,
            dimensionedScalar("rho", dimDensity, rhoRef_)
            )
            );
        }
        else
```

```
        {
                return(obr_.lookupObject<volScalarField>(rhoName_));
        }
}


Foam::scalar Foam::forces::rho(const volScalarField& p) const
{
        if (p.dimensions() == dimPressure)
        {
                return 1.0;
        }
        else
        {
                if (rhoName_ != "rhoInf")
                {
                        FatalErrorIn("forces::rho(const␣volScalarField&␣p)")
                        << "Dynamic␣pressure␣is␣expected␣but␣kinematic␣is␣provided."
                        << exit(FatalError);
                }

                return rhoRef_;
        }
}


void Foam::forces::applyBins
(
const vectorField& Md,
const vectorField& fN,
const vectorField& fT,
const vectorField& fP,
const vectorField& d
)
{
        if (nBin_ == 1)
        {
                force_[0][0] += sum(fN);
                force_[1][0] += sum(fT);
                force_[2][0] += sum(fP);
                moment_[0][0] += sum(Md^fN);
                moment_[1][0] += sum(Md^fT);
                moment_[2][0] += sum(Md^fP);
        }
        else
        {
                scalarField dd((d & binDir_) - binMin_);

                forAll(dd, i)
                {
        label bini = min(max(floor(dd[i]/binDx_), 0), force_[0].size() - 1);

                        force_[0][bini] += fN[i];
                        force_[1][bini] += fT[i];
```

```
                        force_[2][bini] += fP[i];
                        moment_[0][bini] += Md[i]^fN[i];
                        moment_[1][bini] += Md[i]^fT[i];
                        moment_[2][bini] += Md[i]^fP[i];
                }
        }
}


void Foam::forces::writeForces()
{
        if (log_) Info
        << type() << " " << name_ << " output:" << nl
        << "    sum of forces:" << nl
        << "        pressure : " << sum(force_[0]) << nl
        << "        viscous   : " << sum(force_[1]) << nl
        << "        porous    : " << sum(force_[2]) << nl
        << "    sum of moments:" << nl
        << "        pressure : " << sum(moment_[0]) << nl
        << "        viscous   : " << sum(moment_[1]) << nl
        << "        porous    : " << sum(moment_[2])
        << endl;

        file(0) << obr_.time().value() << tab << setw(1) << '('
        << sum(force_[0]) << setw(1) << ' '
        << sum(force_[1]) << setw(1) << ' '
        << sum(force_[2]) << setw(3) << ") ("
        << sum(moment_[0]) << setw(1) << ' '
        << sum(moment_[1]) << setw(1) << ' '
        << sum(moment_[2]) << setw(1) << ')'
        << endl;

        if (localSystem_)
        {
                vectorField localForceN(coordSys_.localVector(force_[0]));
                vectorField localForceT(coordSys_.localVector(force_[1]));
                vectorField localForceP(coordSys_.localVector(force_[2]));
                vectorField localMomentN(coordSys_.localVector(moment_[0]));
                vectorField localMomentT(coordSys_.localVector(moment_[1]));
                vectorField localMomentP(coordSys_.localVector(moment_[2]));

                file(0) << obr_.time().value() << tab << setw(1) << '('
                << sum(localForceN) << setw(1) << ' '
                << sum(localForceT) << setw(1) << ' '
                << sum(localForceP) << setw(3) << ") ("
                << sum(localMomentN) << setw(1) << ' '
                << sum(localMomentT) << setw(1) << ' '
                << sum(localMomentP) << setw(1) << ')'
                << endl;
        }
}


void Foam::forces::writeBins()
```

```
{
        if (nBin_ == 1)
        {
                return;
        }

        List<Field<vector> > f(force_);
        List<Field<vector> > m(moment_);

        if (binCumulative_)
        {
                for (label i = 1; i < f[0].size(); i++)
                {
                        f[0][i] += f[0][i-1];
                        f[1][i] += f[1][i-1];
                        f[2][i] += f[2][i-1];

                        m[0][i] += m[0][i-1];
                        m[1][i] += m[1][i-1];
                        m[2][i] += m[2][i-1];
                }
        }

        file(1) << obr_.time().value();

        forAll(f[0], i)
        {
                file(1)
                << tab << setw(1) << '('
                << f[0][i] << setw(1) << ' '
                << f[1][i] << setw(1) << ' '
                << f[2][i] << setw(3) << ") ("
                << m[0][i] << setw(1) << ' '
                << m[1][i] << setw(1) << ' '
                << m[2][i] << setw(1) << ')';
        }

        if (localSystem_)
        {
                List<Field<vector> > lf(3);
                List<Field<vector> > lm(3);
                lf[0] = coordSys_.localVector(force_[0]);
                lf[1] = coordSys_.localVector(force_[1]);
                lf[2] = coordSys_.localVector(force_[2]);
                lm[0] = coordSys_.localVector(moment_[0]);
                lm[1] = coordSys_.localVector(moment_[1]);
                lm[2] = coordSys_.localVector(moment_[2]);

                if (binCumulative_)
                {
                        for (label i = 1; i < lf[0].size(); i++)
                        {
                                lf[0][i] += lf[0][i-1];
                                lf[1][i] += lf[1][i-1];
```

```
                                lf[2][i] += lf[2][i-1];
                                lm[0][i] += lm[0][i-1];
                                lm[1][i] += lm[1][i-1];
                                lm[2][i] += lm[2][i-1];
                        }
                }

                forAll(lf[0], i)
                {
                        file(1)
                        << tab << setw(1) << '('
                        << lf[0][i] << setw(1) << ' '
                        << lf[1][i] << setw(1) << ' '
                        << lf[2][i] << setw(3) << ") ("
                        << lm[0][i] << setw(1) << ' '
                        << lm[1][i] << setw(1) << ' '
                        << lm[2][i] << setw(1) << ')';
                }
        }

        file(1) << endl;
}


// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * //

Foam::forces::forces
(
const word& name,
const objectRegistry& obr,
const dictionary& dict,
const bool loadFromFiles,
const bool readFields
)
:
functionObjectFile(obr, name, createFileNames(dict)),
name_(name),
obr_(obr),
active_(true),
log_(true),
force_(3),
moment_(3),
patchSet_(),
pName_(word::null),
UName_(word::null),
rhoName_(word::null),
directForceDensity_(false),
fDName_(""),
rhoRef_(VGREAT),
pRef_(0),
coordSys_(),
localSystem_(false),
porosity_(false),
nBin_(1),
```

```
binDir_(vector::zero),
binDx_(0.0),
binMin_(GREAT),
binPoints_(),
binCumulative_(true),
initialised_(false)
{
        // Check if the available mesh is an fvMesh otherise deactivate
        if (isA<fvMesh>(obr_))
        {
                if (readFields)
                {
                        read(dict);
                        Info<< endl;
                }
        }
        else
        {
                active_ = false;
                WarningIn
                (
                "Foam::forces::forces"
                "("
                "const␣word&,␣"
                "const␣objectRegistry&,␣"
                "const␣dictionary&,␣"
                "const␣bool"
                ")"
                )   << "No␣fvMesh␣available,␣deactivating␣" << name_
                << endl;
        }

}


Foam::forces::forces
(
const word& name,
const objectRegistry& obr,
const labelHashSet& patchSet,
const word& pName,
const word& UName,
const word& rhoName,
const scalar rhoInf,
const scalar pRef,
const coordinateSystem& coordSys
)
:
functionObjectFile(obr, name, typeName),
name_(name),
obr_(obr),
active_(true),
log_(true),
force_(3),
```

```
moment_(3),
patchSet_(patchSet),
pName_(pName),
UName_(UName),
rhoName_(rhoName),
directForceDensity_(false),
fDName_(""),
rhoRef_(rhoInf),
pRef_(pRef),
coordSys_(coordSys),
localSystem_(false),
porosity_(false),
nBin_(1),
binDir_(vector::zero),
binDx_(0.0),
binMin_(GREAT),
binPoints_(),
binCumulative_(true),
initialised_(false)
{
        forAll(force_, i)
        {
                force_[i].setSize(nBin_);
                moment_[i].setSize(nBin_);
        }
}


// * * * * * Destructor  * * * *//

Foam::forces::~forces()
{}


// * * * * * Member Functions  * * * *//

void Foam::forces::read(const dictionary& dict)
{
        if (active_)
        {
                initialised_ = false;

                log_ = dict.lookupOrDefault<Switch>("log", false);

                if (log_) Info<< type() << "␣" << name_ << ":" << nl;

                directForceDensity_ = dict.lookupOrDefault("directForceDensity", false);

                const fvMesh& mesh = refCast<const fvMesh>(obr_);
                const polyBoundaryMesh& pbm = mesh.boundaryMesh();

                patchSet_ = pbm.patchSet(wordReList(dict.lookup("patches")));

                if (directForceDensity_)
```

```
{
        // Optional entry for fDName
        fDName_ = dict.lookupOrDefault<word>("fDName", "fD");
}
else
{
        // Optional entries U and p
        pName_ = dict.lookupOrDefault<word>("pName", "p");
        UName_ = dict.lookupOrDefault<word>("UName", "U");
        rhoName_ = dict.lookupOrDefault<word>("rhoName", "rho");

        // Reference density needed for incompressible calculations
        rhoRef_ = readScalar(dict.lookup("rhoInf"));

        // Reference pressure, 0 by default
        pRef_ = dict.lookupOrDefault<scalar>("pRef", 0.0);
}

coordSys_.clear();

// Centre of rotation for moment calculations
// specified directly, from coordinate system, or implicitly (0 0 0)
if (!dict.readIfPresent<point>("CofR", coordSys_.origin()))
{
        coordSys_ = coordinateSystem(obr_, dict);
        localSystem_ = true;
}

dict.readIfPresent("porosity", porosity_);
if (porosity_)
{
        if (log_) Info<< "    Including porosity effects" << endl;
}
else
{
        if (log_) Info<< "    Not including porosity effects" << endl;
}

if (dict.found("binData"))
{
        const dictionary& binDict(dict.subDict("binData"));
        binDict.lookup("nBin") >> nBin_;

        if (nBin_ < 0)
        {
                FatalIOErrorIn
                (
                "void Foam::forces::read(const dictionary&)", dict
                )   << "Number of bins (nBin) must be zero or greater"
                << exit(FatalIOError);
        }
        else if ((nBin_ == 0) || (nBin_ == 1))
        {
                nBin_ = 1;
```

```
                        forAll(force_, i)
                        {
                                force_[i].setSize(1);
                                moment_[i].setSize(1);
                        }
                }

                if (nBin_ > 1)
                {
                        binDict.lookup("direction") >> binDir_;
                        binDir_ /= mag(binDir_);

                        binMin_ = GREAT;
                        scalar binMax = -GREAT;
                        forAllConstIter(labelHashSet, patchSet_, iter)
                        {
                                label patchI = iter.key();
                                const polyPatch& pp = pbm[patchI];
                                scalarField d(pp.faceCentres() & binDir_);
                                binMin_ = min(min(d), binMin_);
                                binMax = max(max(d), binMax);
                        }
                        reduce(binMin_, minOp<scalar>());
                        reduce(binMax, maxOp<scalar>());

    // slightly boost binMax so that region of interest is fully
// within bounds
                        binMax = 1.0001*(binMax - binMin_) + binMin_;

                        binDx_ = (binMax - binMin_)/scalar(nBin_);

                        // create the bin points used for writing
                        binPoints_.setSize(nBin_);
                        forAll(binPoints_, i)
                        {
                                binPoints_[i] = (i + 0.5)*binDir_*binDx_;
                        }

                        binDict.lookup("cumulative") >> binCumulative_;

                        // allocate storage for forces and moments
                        forAll(force_, i)
                        {
                                force_[i].setSize(nBin_);
                                moment_[i].setSize(nBin_);
                        }
                }
        }

        if (nBin_ == 1)
        {
                // allocate storage for forces and moments
                force_[0].setSize(1);
                force_[1].setSize(1);
```

```
                        force_[2].setSize(1);
                        moment_[0].setSize(1);
                        moment_[1].setSize(1);
                        moment_[2].setSize(1);
                }
        }
}


void Foam::forces::execute()
{
        // Do nothing - only valid on write
}


void Foam::forces::end()
{
        // Do nothing - only valid on write
}


void Foam::forces::timeSet()
{
        // Do nothing - only valid on write
}


void Foam::forces::write()
{
        calcForcesMoment();

        if (!active_)
        {
                return;
        }

        if (Pstream::master())
        {
                functionObjectFile::write();

                writeForces();

                writeBins();

                if (log_) Info<< endl;
        }
}


void Foam::forces::calcForcesMoment()
{
        initialise();

        if (!active_)
```

```
    {
        return;
    }

    force_[0] = vector::zero;
    force_[1] = vector::zero;
    force_[2] = vector::zero;

    moment_[0] = vector::zero;
    moment_[1] = vector::zero;
    moment_[2] = vector::zero;

    if (directForceDensity_)
    {
        const volVectorField& fD = obr_.lookupObject<volVectorField>(fDName_);

        const fvMesh& mesh = fD.mesh();

        const surfaceVectorField::GeometricBoundaryField& Sfb =
        mesh.Sf().boundaryField();

        forAllConstIter(labelHashSet, patchSet_, iter)
        {
            label patchI = iter.key();

            vectorField Md
            (
            mesh.C().boundaryField()[patchI] - coordSys_.origin()
            );

            scalarField sA(mag(Sfb[patchI]));

    // Normal force = surfaceUnitNormal*(surfaceNormal & forceDensity)
            vectorField fN
            (
            Sfb[patchI]/sA
            *(
            Sfb[patchI] & fD.boundaryField()[patchI]
            )
            );

// Tangential force (total force minus normal fN)
            vectorField fT(sA*fD.boundaryField()[patchI] - fN);

    //- Porous force
            vectorField fP(Md.size(), vector::zero);

            applyBins(Md, fN, fT, fP, mesh.C().boundaryField()[patchI]);
        }
    }
    else
    {
        const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);
        const volScalarField& p = obr_.lookupObject<volScalarField>(pName_);
```

```
        const fvMesh& mesh = U.mesh();

        const surfaceVectorField::GeometricBoundaryField& Sfb =
        mesh.Sf().boundaryField();

        tmp<volSymmTensorField> tdevRhoReff = devRhoReff();
        const volSymmTensorField::GeometricBoundaryField& devRhoReffb
        = tdevRhoReff().boundaryField();

        // Scale pRef by density for incompressible simulations
        scalar pRef = pRef_/rho(p);

        forAllConstIter(labelHashSet, patchSet_, iter)
        {
                label patchI = iter.key();

                vectorField Md
                (
                mesh.C().boundaryField()[patchI] - coordSys_.origin()
                );

                vectorField fN
                (
                rho(p)*Sfb[patchI]*(p.boundaryField()[patchI] - pRef)
                );

                vectorField fT(Sfb[patchI] & devRhoReffb[patchI]);

                vectorField fP(Md.size(), vector::zero);

                applyBins(Md, fN, fT, fP, mesh.C().boundaryField()[patchI]);
        }
}

if (porosity_)
{
        const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);
        const volScalarField rho(this->rho());
        const volScalarField mu(this->mu());

        const fvMesh& mesh = U.mesh();

        const HashTable<const porosityModel*> models =
        obr_.lookupClass<porosityModel>();

        if (models.empty())
        {
                WarningIn("void␣Foam::forces::calcForcesMoment()")
                << "Porosity␣effects␣requested,␣but␣no␣porosity␣models␣found␣"
                << "in␣the␣database"
                << endl;
        }
```

```
                forAllConstIter(HashTable<const porosityModel*>, models, iter)
                {
                        // non-const access required if mesh is changing
                        porosityModel& pm = const_cast<porosityModel&>(*iter());

                        vectorField fPTot(pm.force(U, rho, mu));

                        const labelList& cellZoneIDs = pm.cellZoneIDs();

                        forAll(cellZoneIDs, i)
                        {
                                label zoneI = cellZoneIDs[i];
                                const cellZone& cZone = mesh.cellZones()[zoneI];

                                const vectorField d(mesh.C(), cZone);
                                const vectorField fP(fPTot, cZone);
                                const vectorField Md(d - coordSys_.origin());

                                const vectorField fDummy(Md.size(), vector::zero);

                                applyBins(Md, fDummy, fDummy, fP, d);
                        }
                }
        }

        Pstream::listCombineGather(force_, plusEqOp<vectorField>());
        Pstream::listCombineGather(moment_, plusEqOp<vectorField>());
        Pstream::listCombineScatter(force_);
        Pstream::listCombineScatter(moment_);
}


Foam::vector Foam::forces::forceEff() const
{
        return sum(force_[0]) + sum(force_[1]) + sum(force_[2]);
}


Foam::vector Foam::forces::momentEff() const
{
        return sum(moment_[0]) + sum(moment_[1]) + sum(moment_[2]);
}


// ************************************************************************* //
```